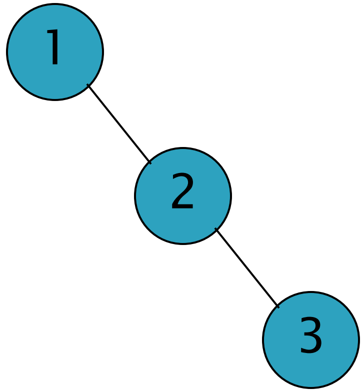
AVL Trees

CS212:Data Structure

AVL Trees

- Consider a situation when data elements are inserted in a BST in sorted order: 1, 2, 3, …



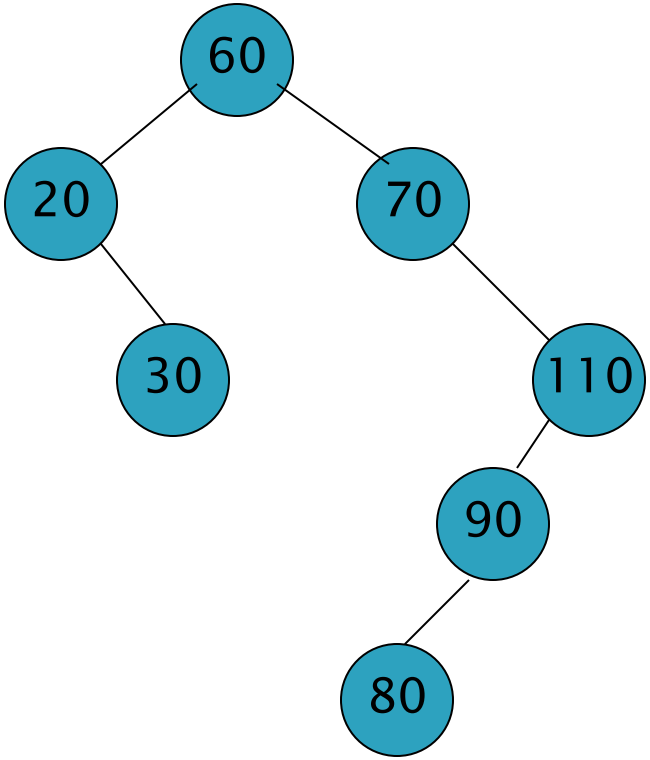
- BST becomes a degenerate tree.
- Search operation **FindKey** takes **O(n)**, which is as inefficient as in a list.

AVL Trees

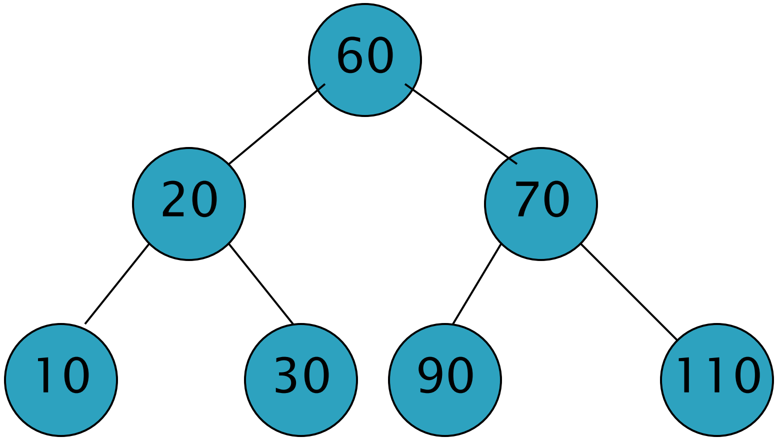
- It is possible that after a number of insert and delete operations a binary tree may become imbalanced and increase in height.
- Can we insert and delete elements from BST so that its height is guaranteed to be **O(logn)**?
  - Yes, AVL Tree ensures this.
- Named after its two inventors: Adelson-Velski and Landis.

Imbalanced/Balanced Trees

An Imbalanced Tree



A Balanced Tree

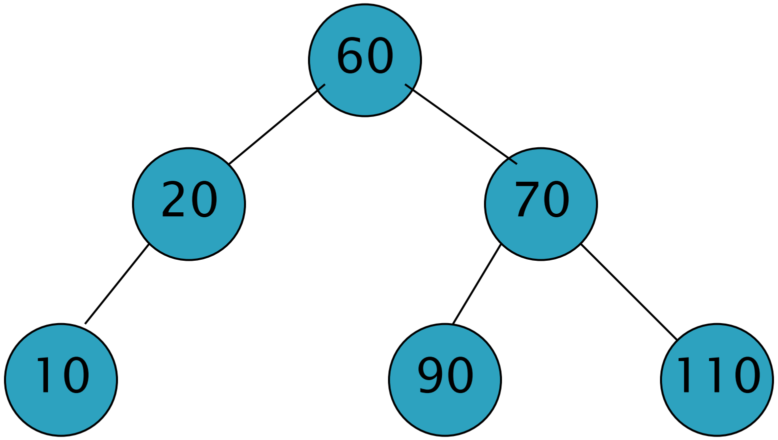


AVL Tree: Definition

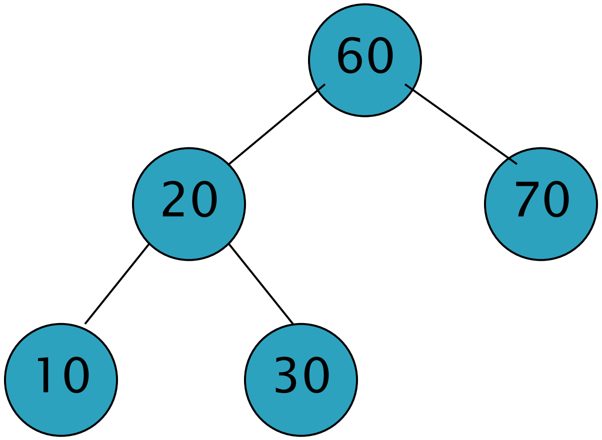
- We cannot always guarantee **perfectly** balanced trees, since this depends on the currently inserted nodes.
- But some nodes arrangements make a tree more balanced than other nodes arrangements.

Imbalanced/Balanced Trees

Balanced Tree?

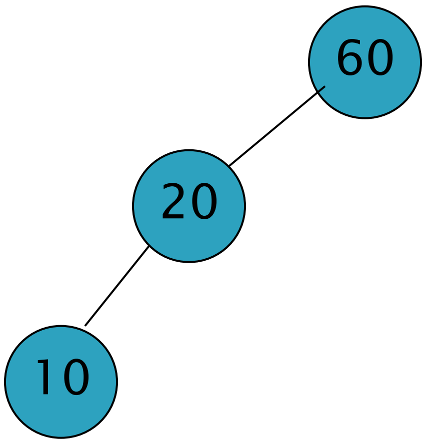


Balanced Tree?

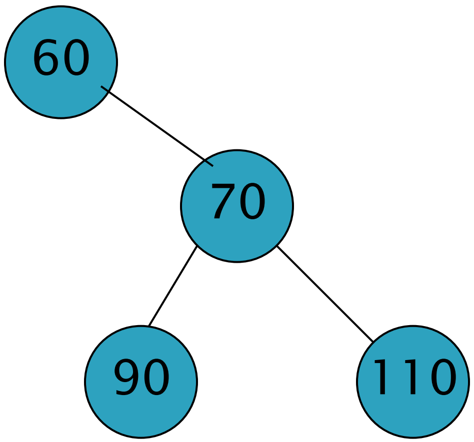


Imbalanced/Balanced Trees

Balanced Tree?

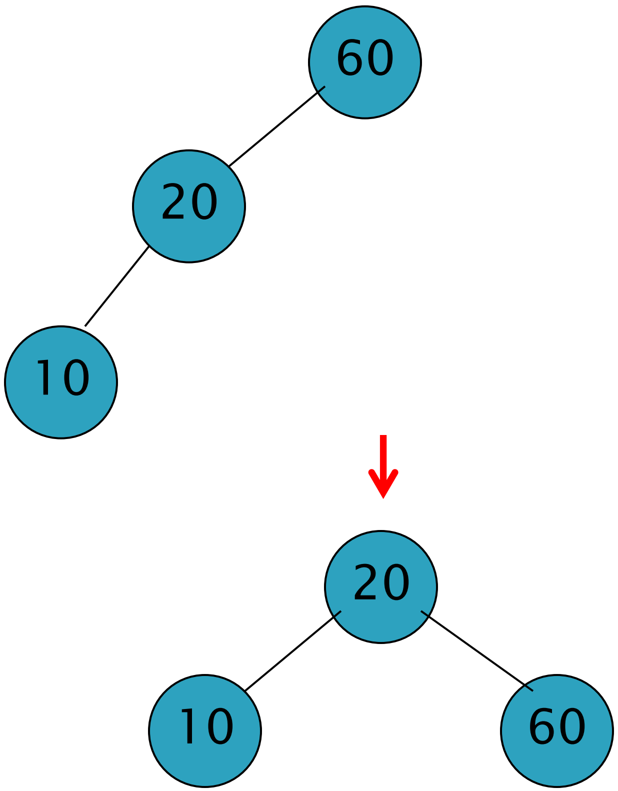


Balanced Tree?

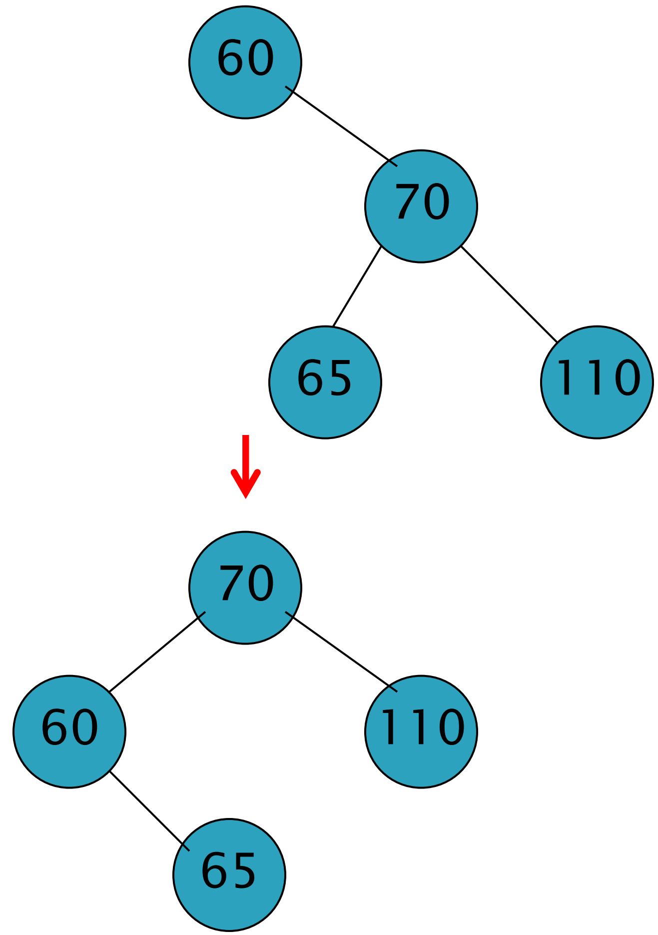


Imbalanced/Balanced Trees

Balanced Tree?



Balanced Tree?

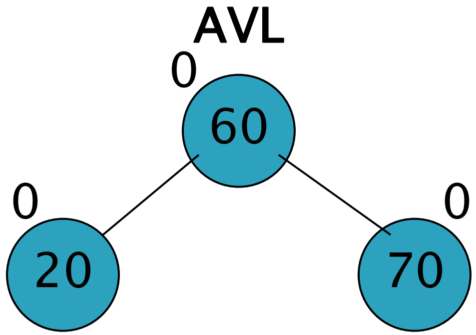


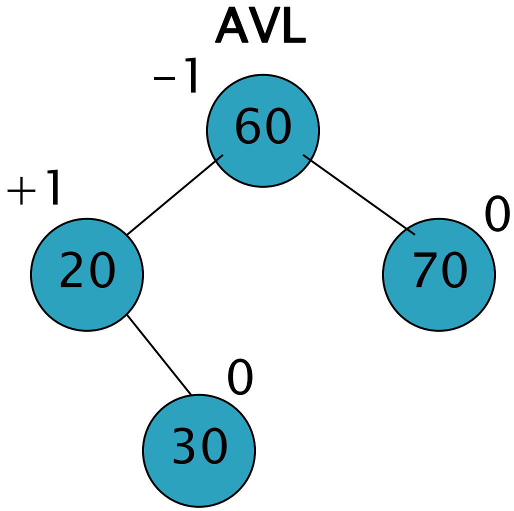
AVL Tree: Definition

- **Height**: the longest path from a node to a leaf node.
- **Height-balanced tree**: A binary tree is a height-balanced-p-tree if for each node in the tree, the absolute difference in height of its two subtrees is at most p.

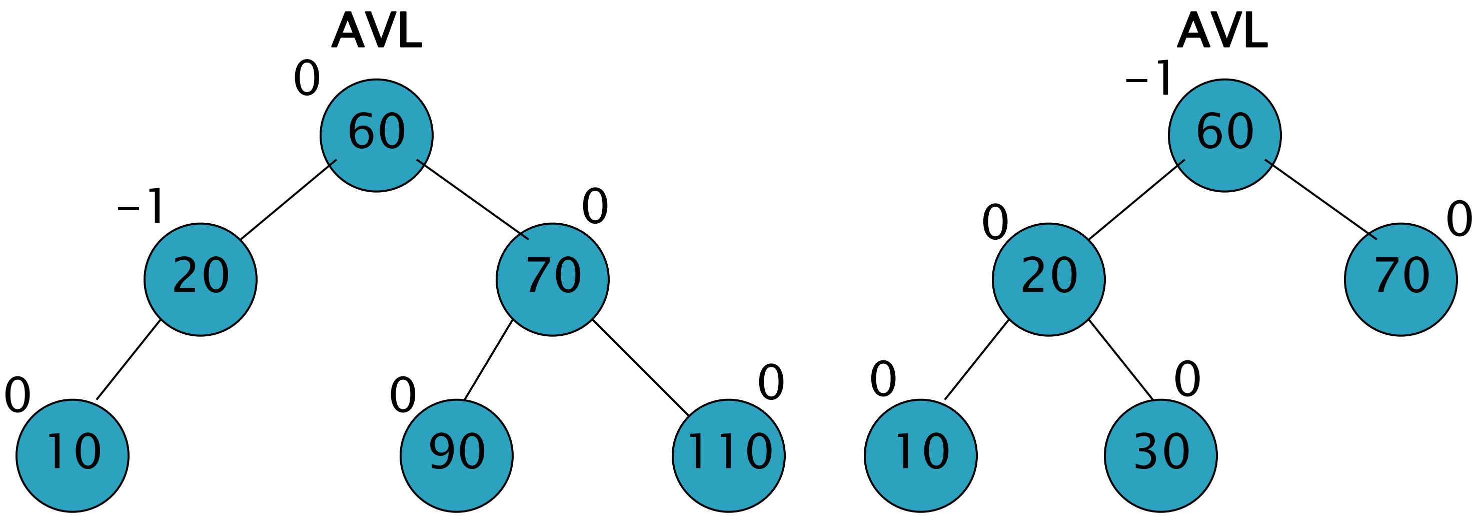
- AVL tree is a **BST** that is **height-balanced-1-tree.**
  - For each node in the tree, the absolute difference in height of its two subtrees must be at most 1.
  - **Balance = Right Subtree Height – Left Subtree Height**
  - Therefore, it must be either **+1** (longer right), **0** (equal), **-1** (longer left).

AVL Trees

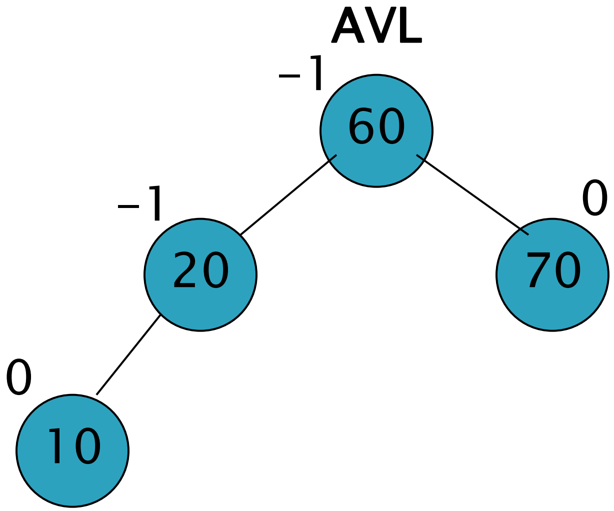


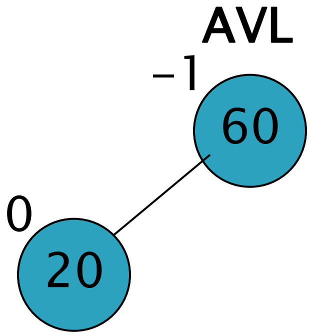


AVL Trees

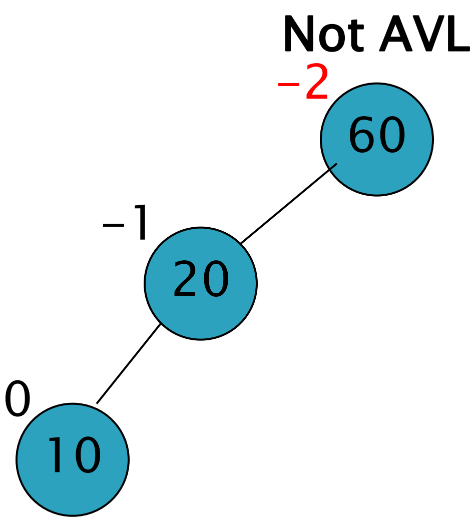


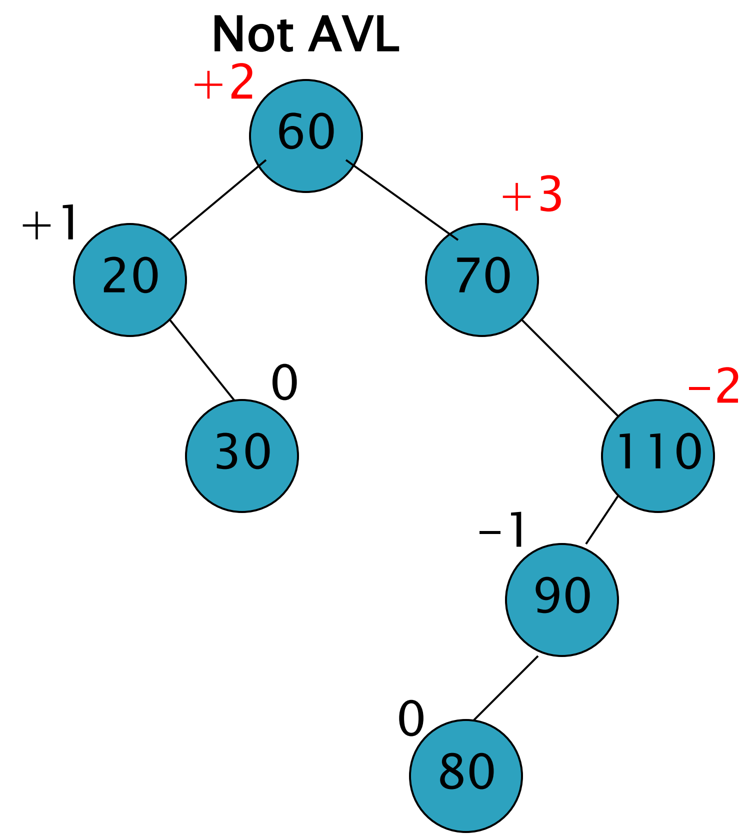
AVL Trees



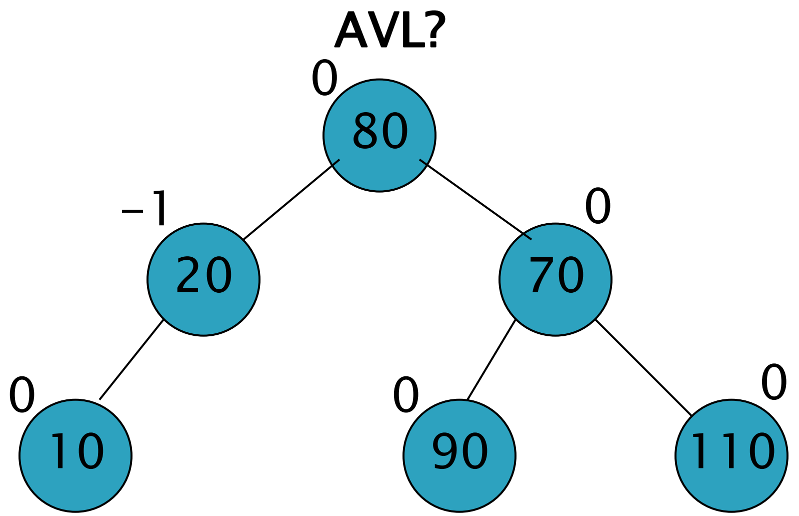


AVL Trees

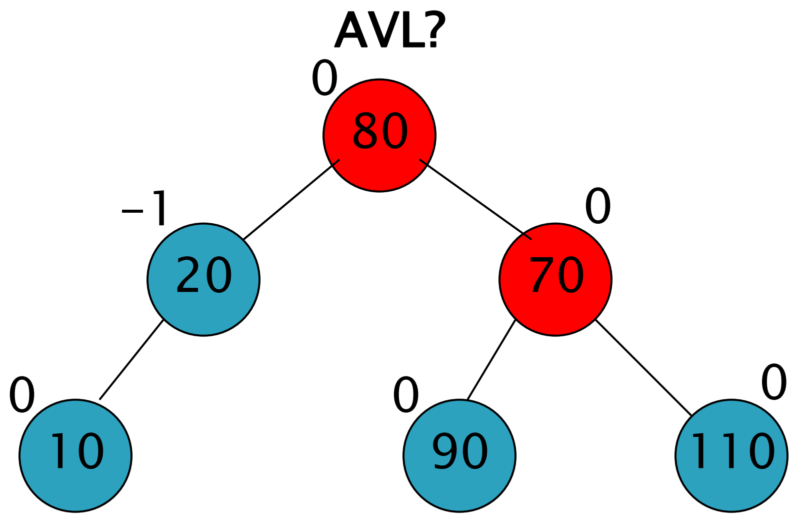




AVL Trees



AVL Trees



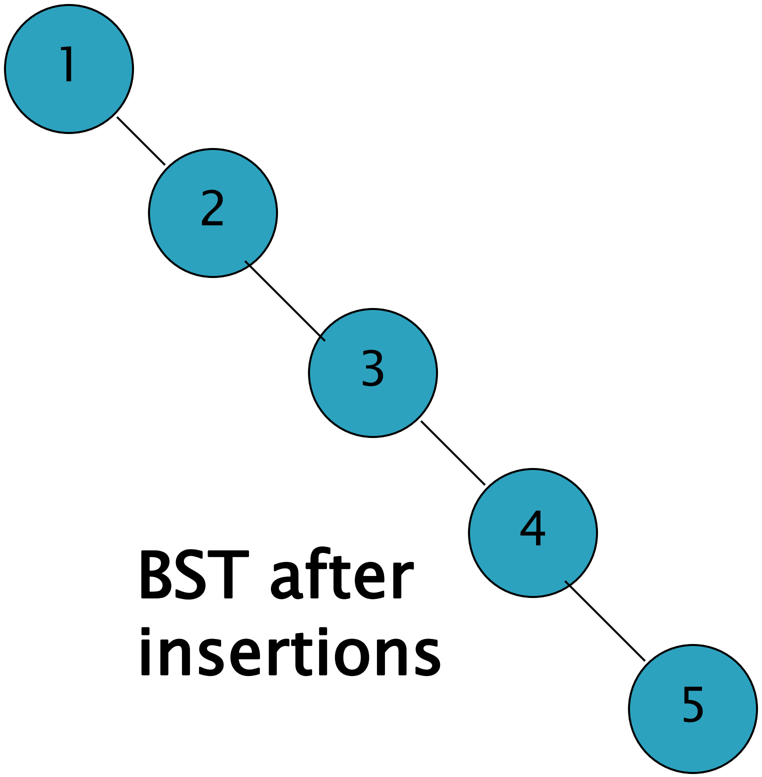
It is balanced tree
but not AVL
because it is not BST!

Remember:

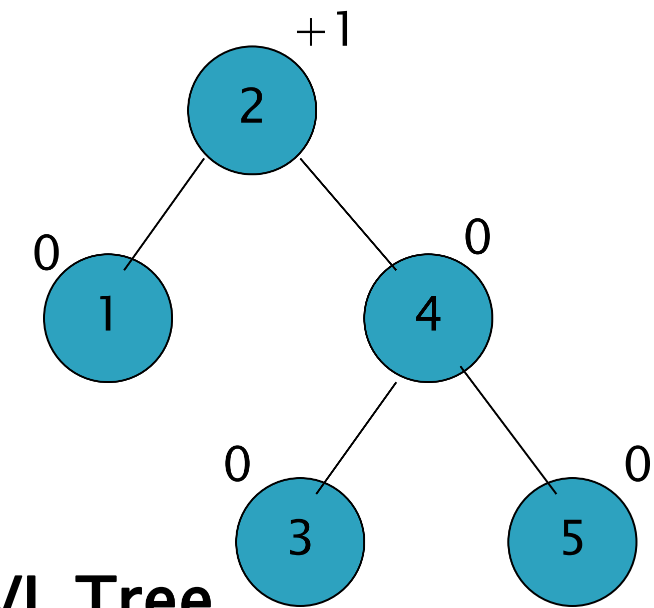
AVL tree is a **BST** that is **height-balanced-1-tree.**

BSTs vs. AVL Trees

Inserting 1, 2, 3, 4 and 5



BST after insertions



AVL Tree after insertions

ADT AVL Tree: Specification

Elements: The elements are nodes, each node contains the following data type: Type.

Structure: Same as for the BST; in addition the height difference of the two subtrees of any node is at the most one.

Domain: the number of nodes in a AVL is bounded; type AVLTree.

ADT AVL Tree: Specification

<u>Operations:</u>

1. **Method** FindKey (int tkey, boolean found).
2. **Method** Insert (int k, Type e, boolean inserted).
3. **Method** Remove_Key (int tkey, boolean deleted)
4. **Method** Update(Type e)
5. **Method** Traverse (Order ord)
6. **Method** DeleteSub ( )
7. **Method** Retrieve (Type e)
8. **Method** Empty (boolean empty).
9. **Method** Full (boolean full)

ADT AVL Tree: Element

```
public class AVLNode<T> {
  public int key
  public T data;
  public Balance bal; // Balance is enum (+1, 0, -1)
  public AVLNode<T> left, right;

  public AVLNode(int key, T data) {
       this.key = key;
       this.data = data;
       bal = Balance.Zero;
       left = right = null;
  }
  ...
  ...
}
```

ADT AVL Tree: Implementation

- The implementation of: **FindKey**, **Update data**, **Traverse**, **Retrieve**, **Empty**, **Full**, and any other method that doesn’t change the tree are exactly like the implementation of BST.
- The only difference in implementation is when we change the nodes of the tree, i.e. **Insert**/**Remove** from the tree.

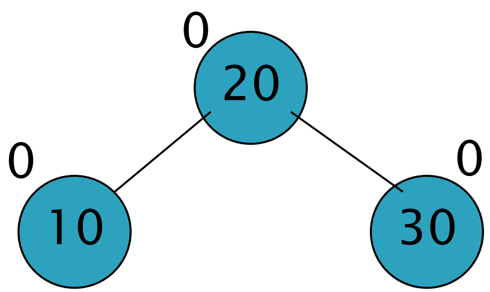
AVL Tree: Insert

- Step 1:
  A node is first inserted into the tree as in a BST.
- Step 2:
  Nodes in the search path are examined to see if there is a pivot node. Three cases arise.
  - search path is a unique path from the root to the new node.
  - pivot node is a node closest to the new node on the search path, whose balance is either –1 or +1.

AVL Tree: Insert

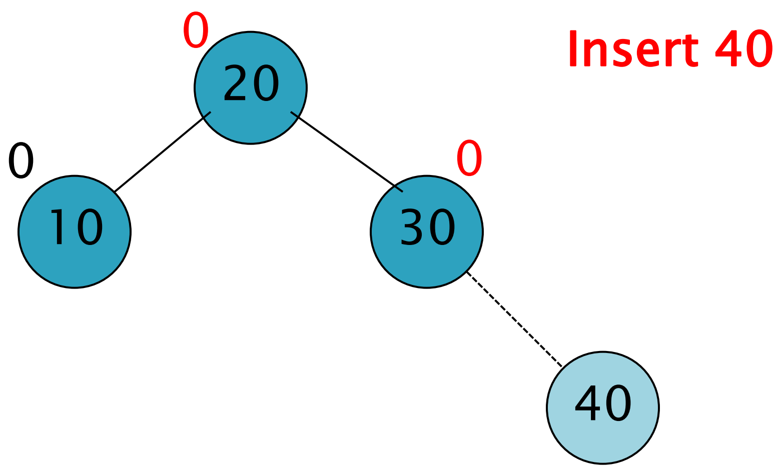
- **Case 1:**
  There is no pivot node in the search path. No adjustment required.
- **Case 2:**
  The pivot node exists and the subtree to which the new node is added has smaller height. No adjustment required.
- **Case 3:**
  The pivot node exists and the subtree to which the new node is added has the larger height. **Adjustment required.**

AVL Tree: Insert (Case 1)

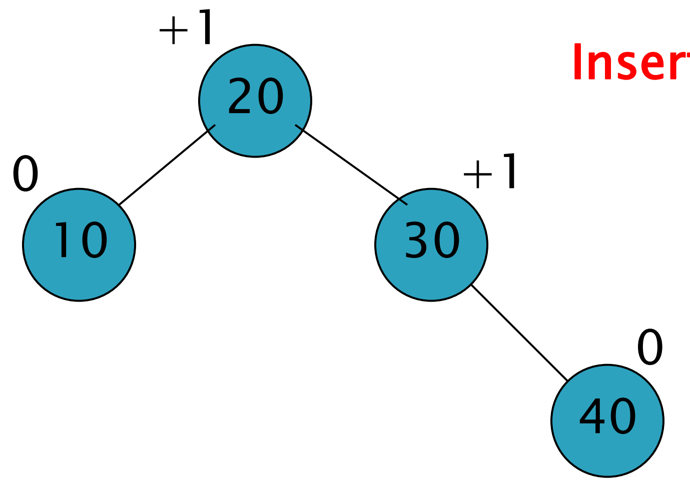


Insert 40

AVL Tree: Insert (Case 1)

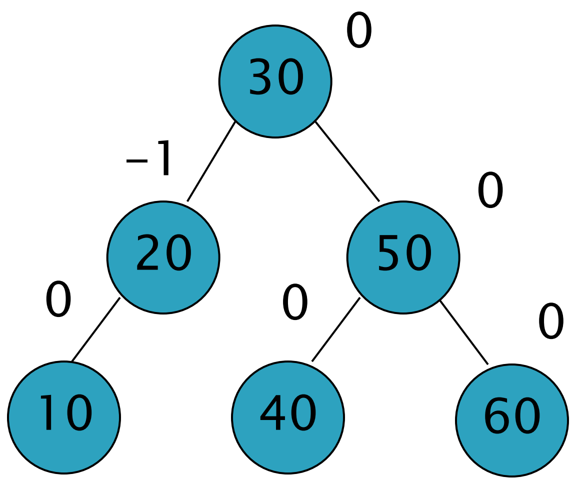


AVL Tree: Insert (Case 1)



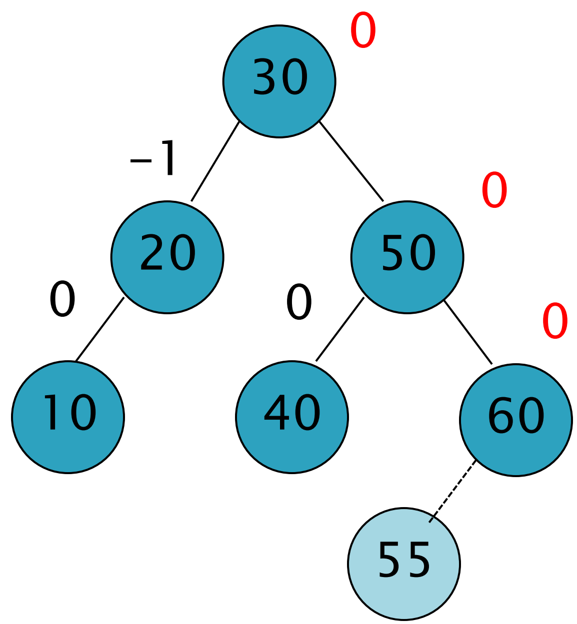
Insert 40

AVL Tree: Insert (Case 1)



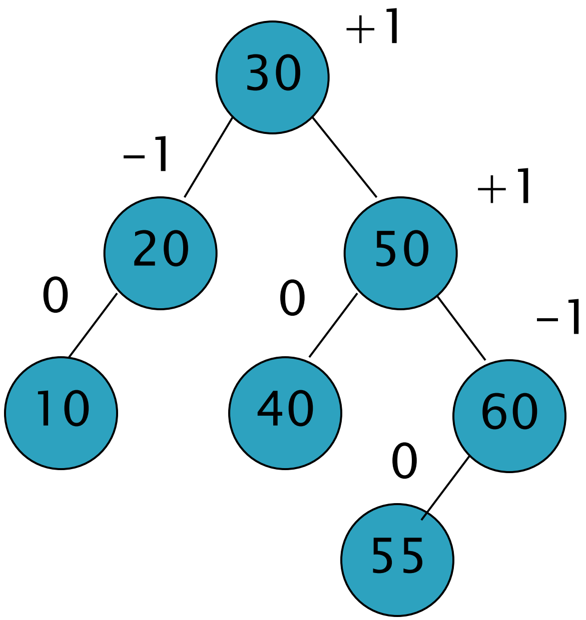
Insert 55

AVL Tree: Insert (Case 1)



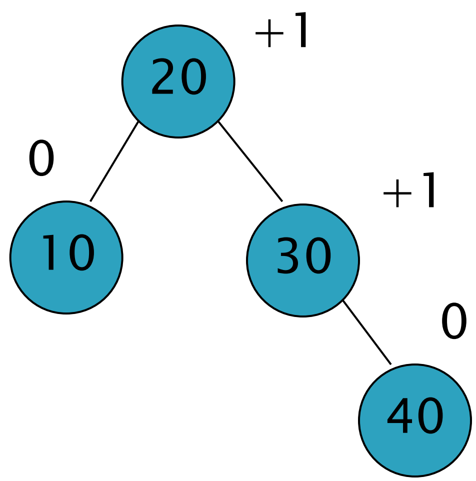
Insert 55

AVL Tree: Insert (Case 1)



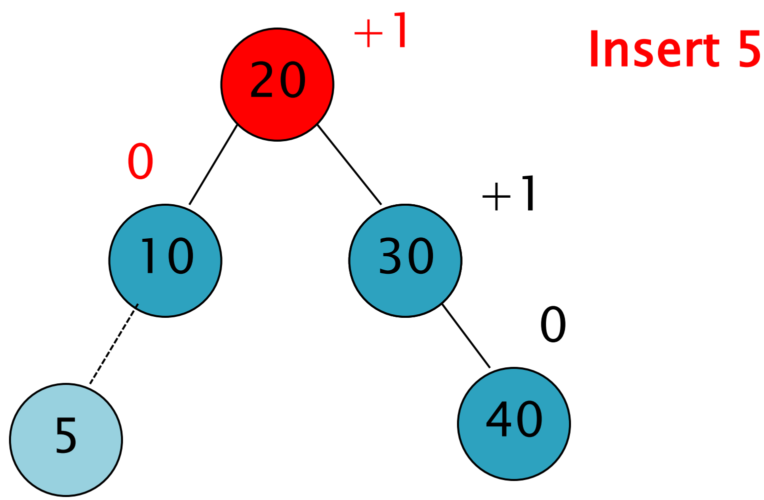
Insert 55

AVL Tree: Insert (Case 2)

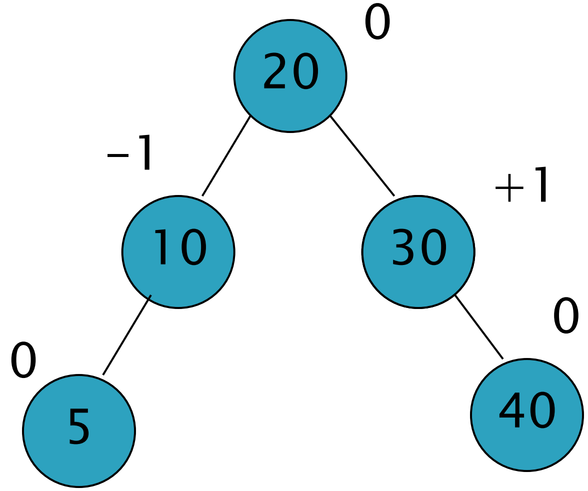


Insert 5

AVL Tree: Insert (Case 2)

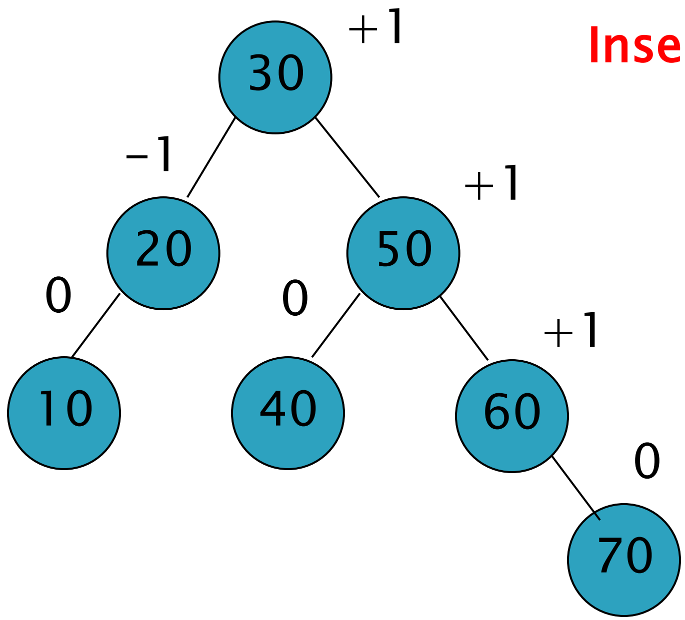


AVL Tree: Insert (Case 2)



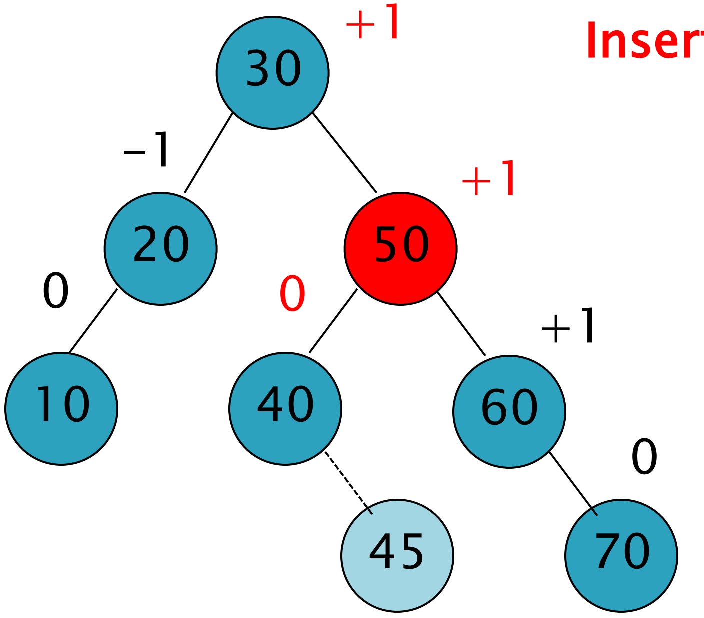
Insert 5

AVL Tree: Insert (Case 2)

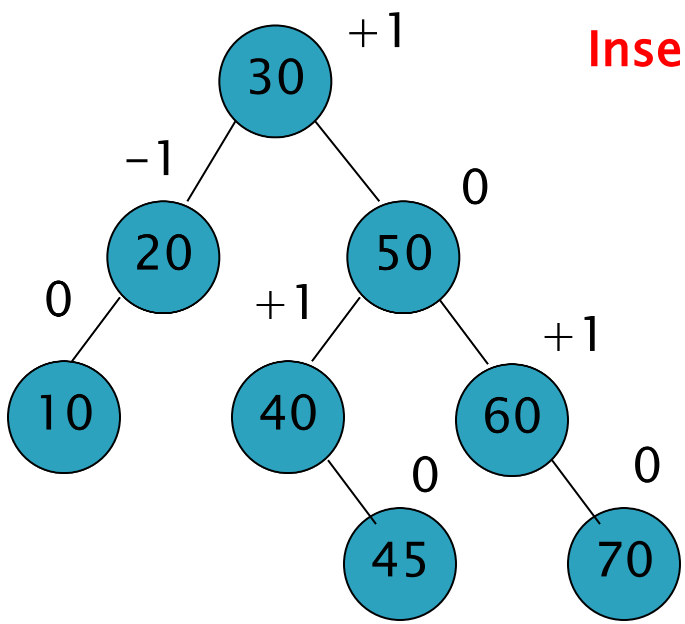


Insert 45

AVL Tree: Insert (Case 2)

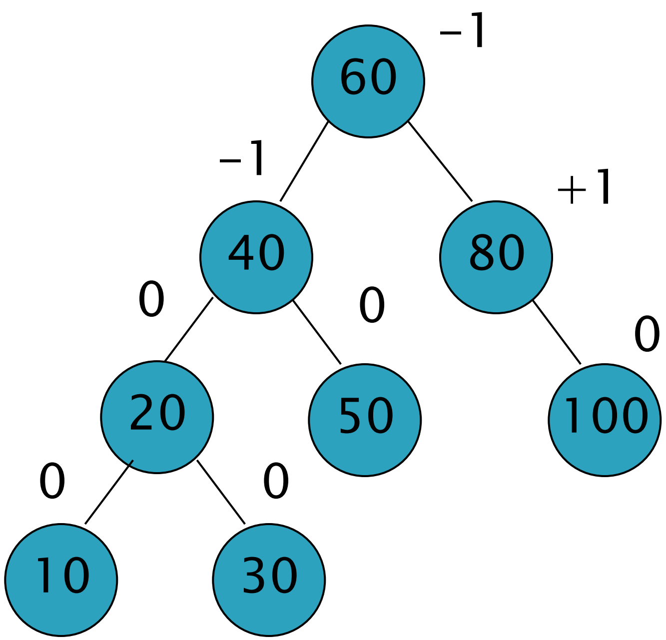


AVL Tree: Insert (Case 2)



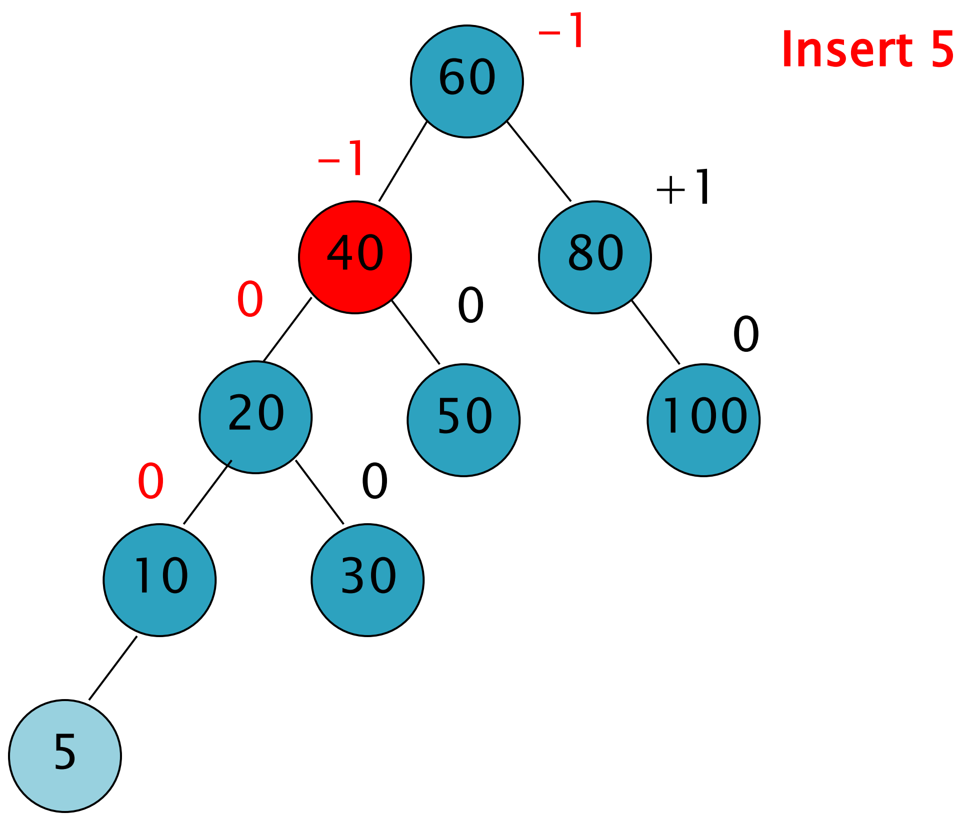
Insert 45

AVL Tree: Insert (Case 3)

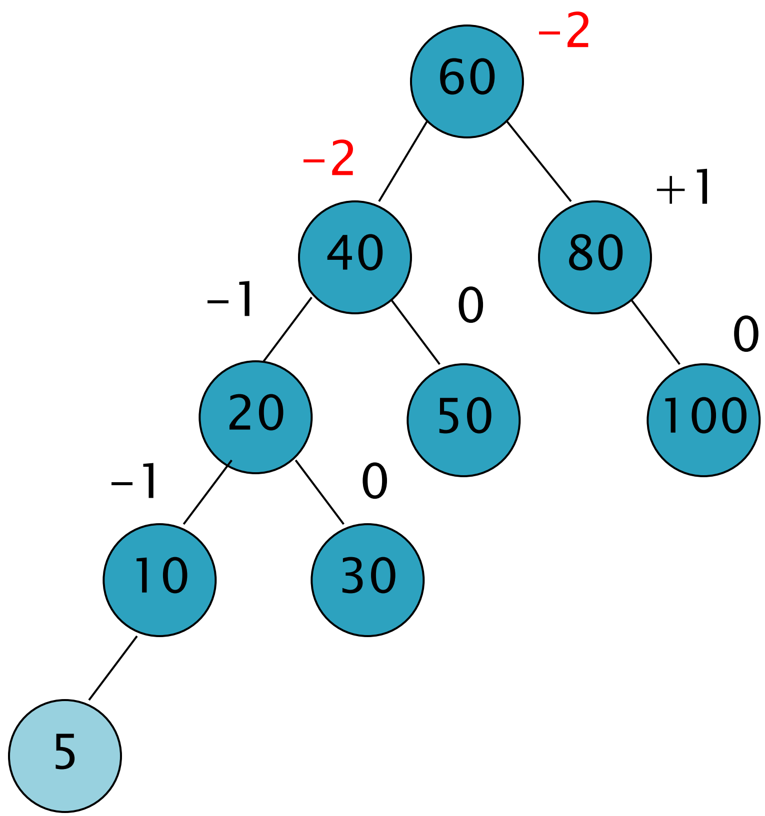


Insert 5

AVL Tree: Insert (Case 3)



AVL Tree: Insert (Case 3)



Insert 5

AVL Tree is no more an AVL Tree after insertion.

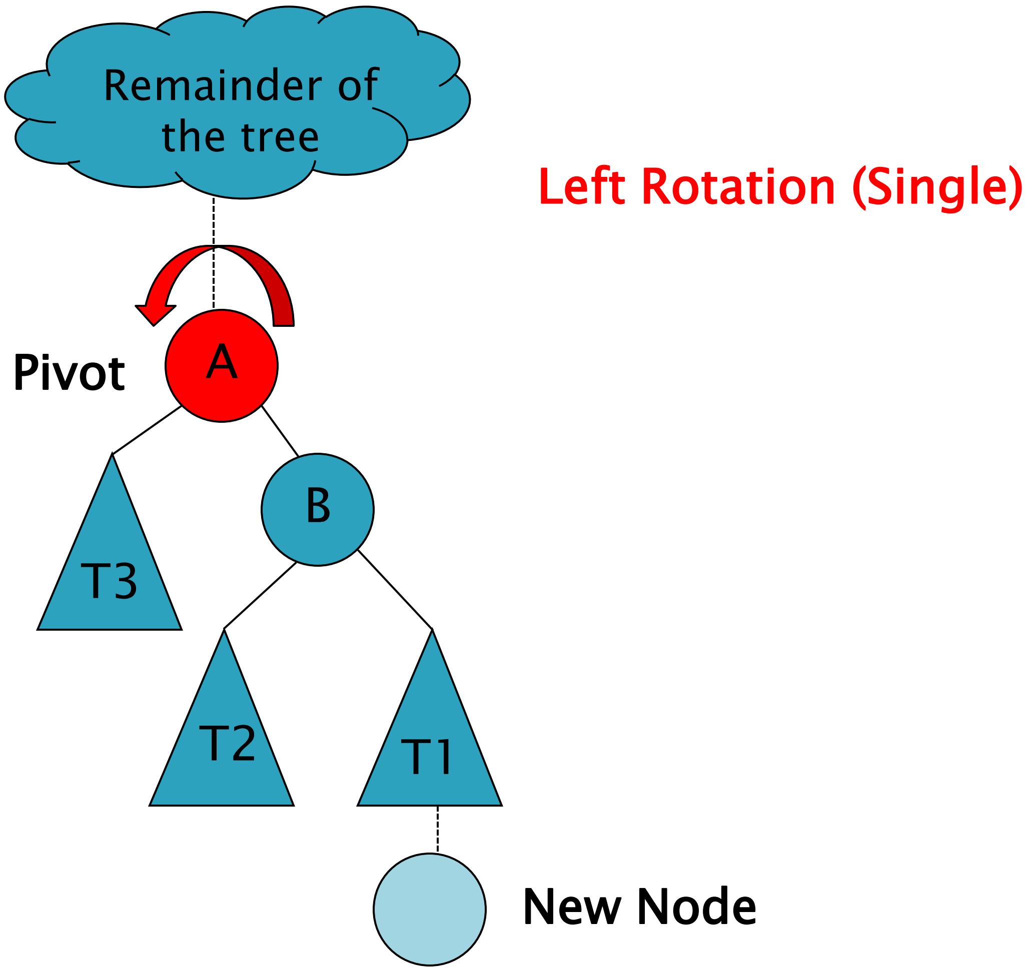
AVL Tree: Insert (Case 3)

- When after an insertion or a deletion an AVL tree becomes imbalanced, adjustments must be made to the tree to change it back into an AVL tree.
- These adjustments are called <u>rotations</u>.
- Rotations can be in the <u>left</u> or <u>right</u> direction.
- Rotations are either <u>single</u> or <u>double</u> rotations.

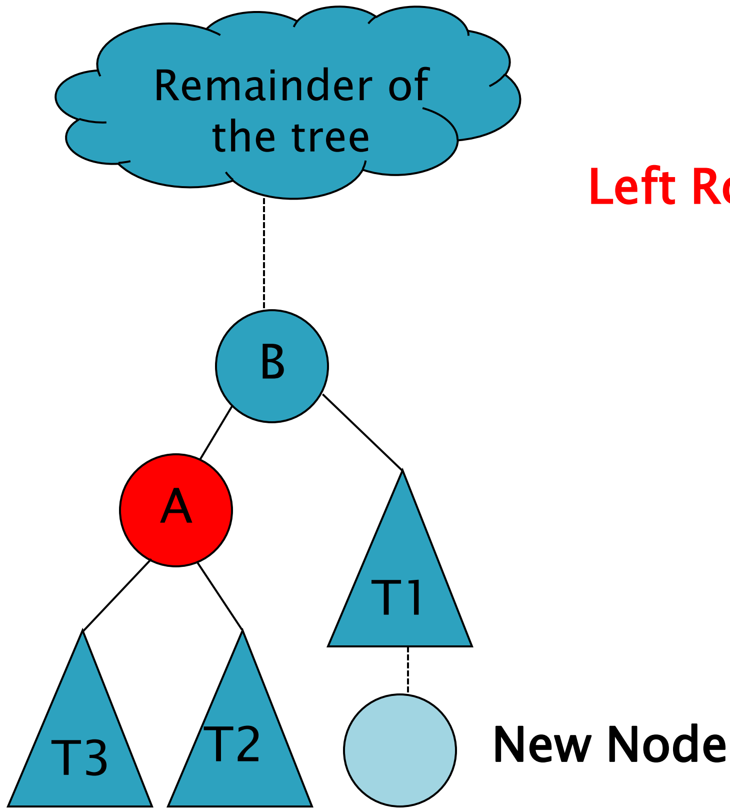
AVL Tree: Insert (Case 3)

- Therefore, there are four different rotations:
  - Left Rotation (Single)
  - Right Rotation (Single)
  - Left-Right Rotations (Double)
  - Right-Left Rotations (Double)

AVL Tree: Insert (Case 3)

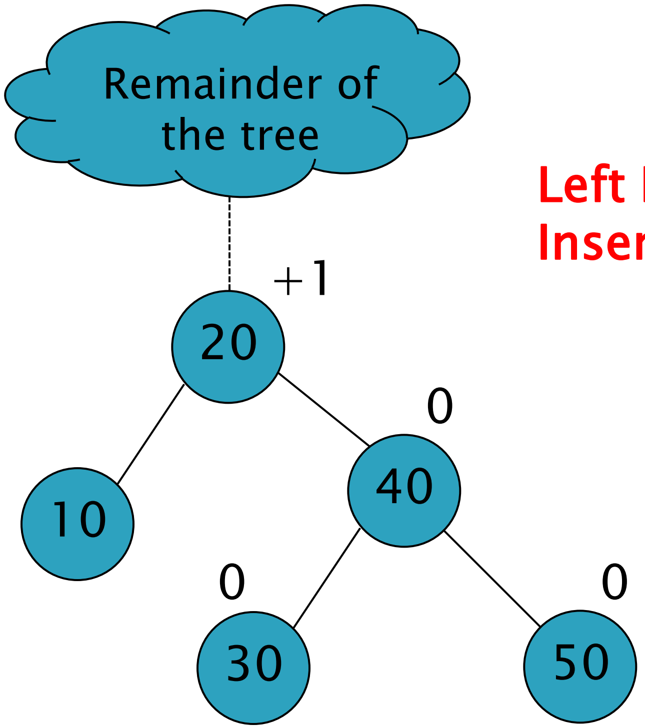


AVL Tree: Insert (Case 3)



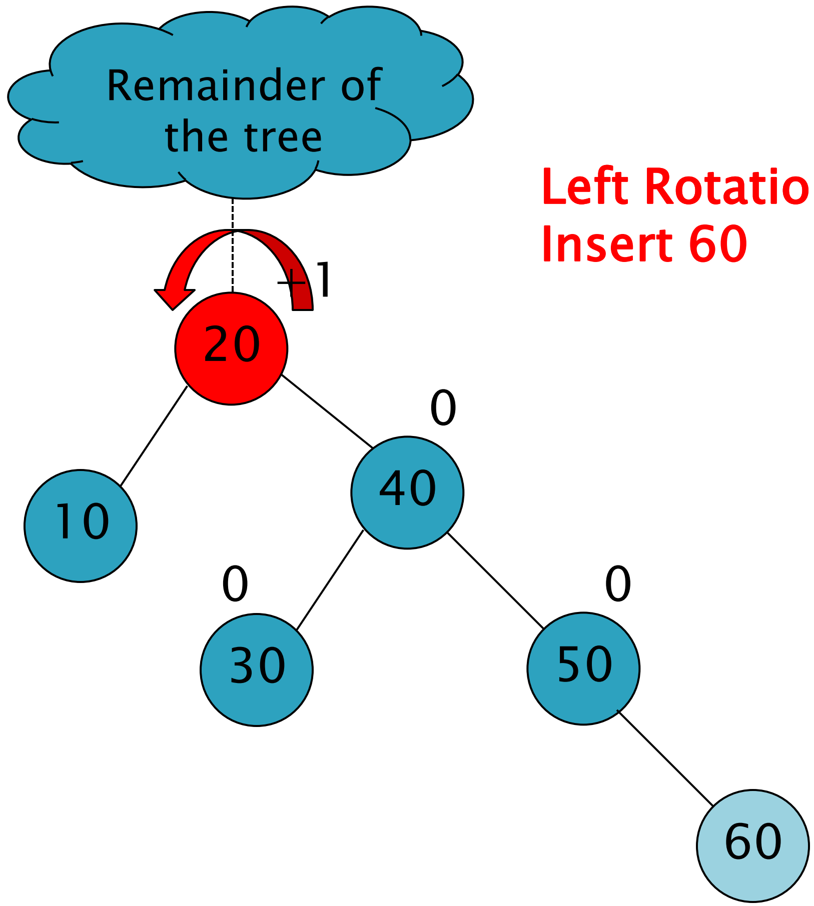
Left Rotation (Single)

AVL Tree: Insert (Case 3)



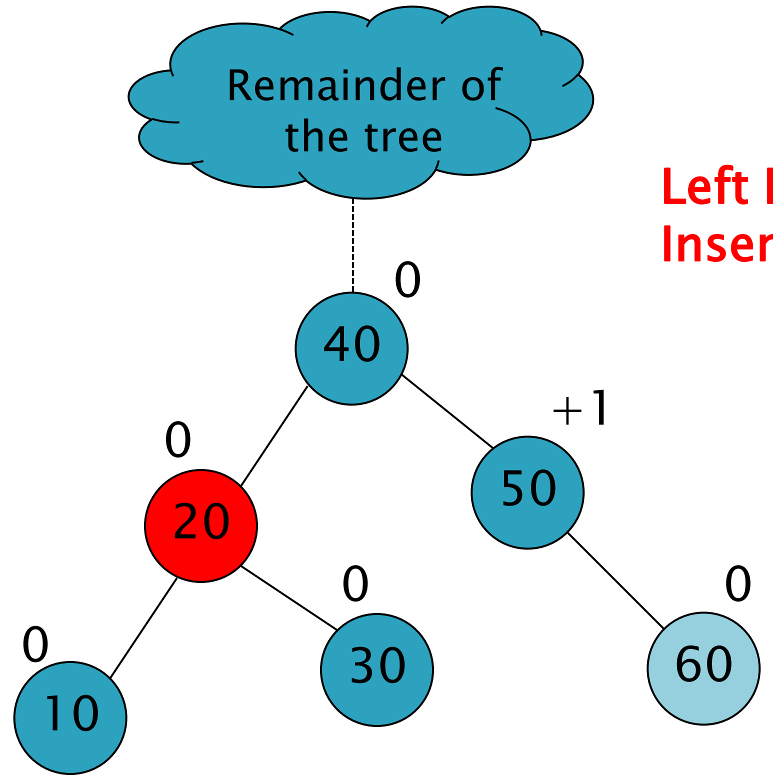
Left Rotation (Single)
Insert 60

AVL Tree: Insert (Case 3)



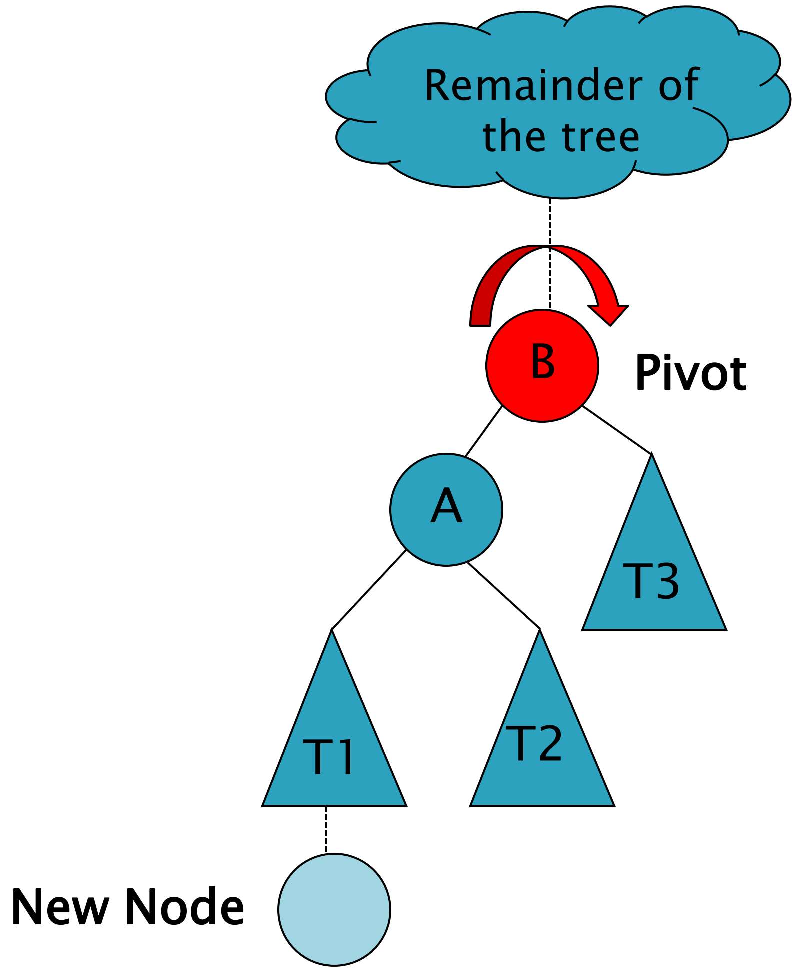
Left Rotation (Single)
Insert 60

AVL Tree: Insert (Case 3)



Left Rotation (Single)
Insert 60

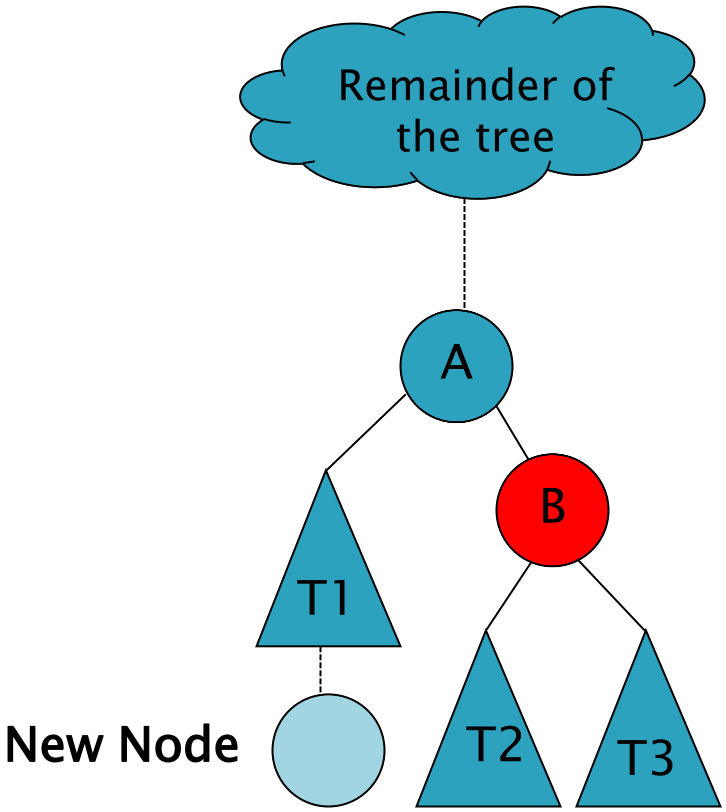
AVL Tree: Insert (Case 3)



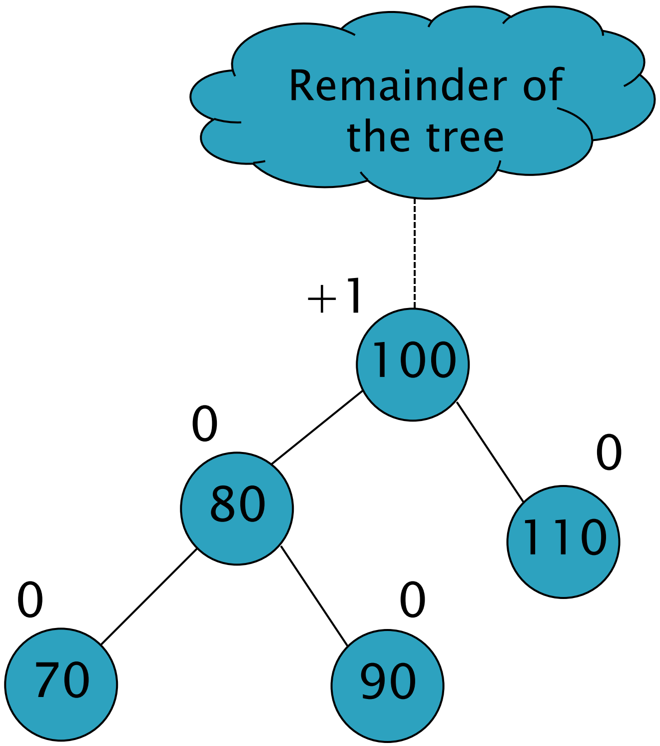
Right Rotation (Single)

AVL Tree: Insert (Case 3)

Right Rotation (Single)

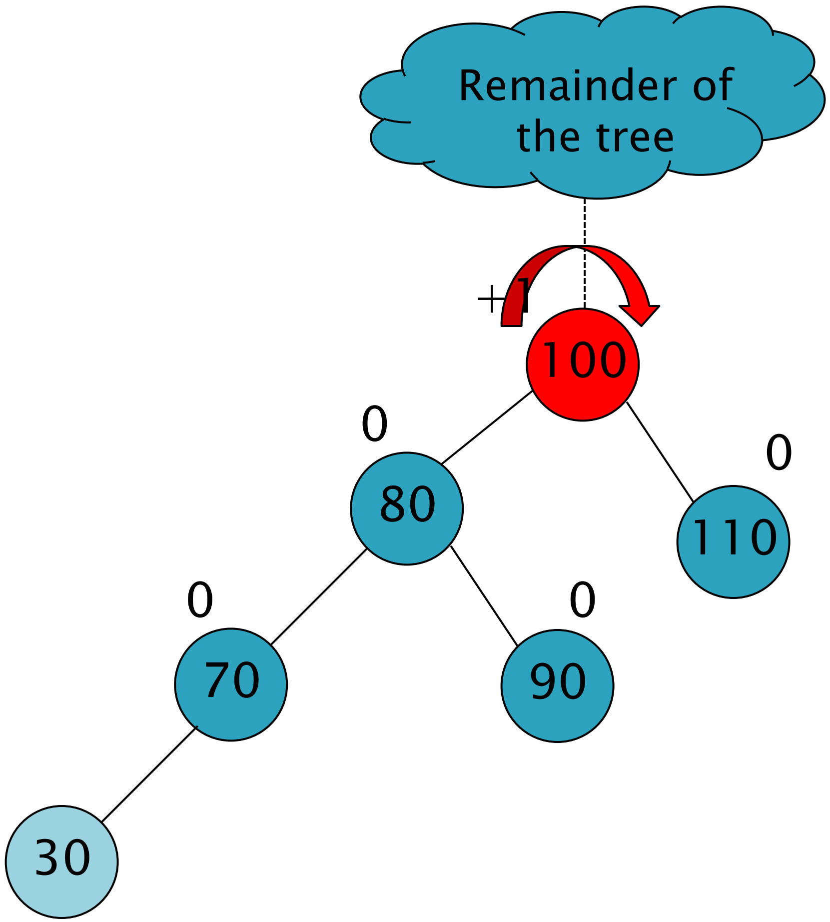


AVL Tree: Insert (Case 3)



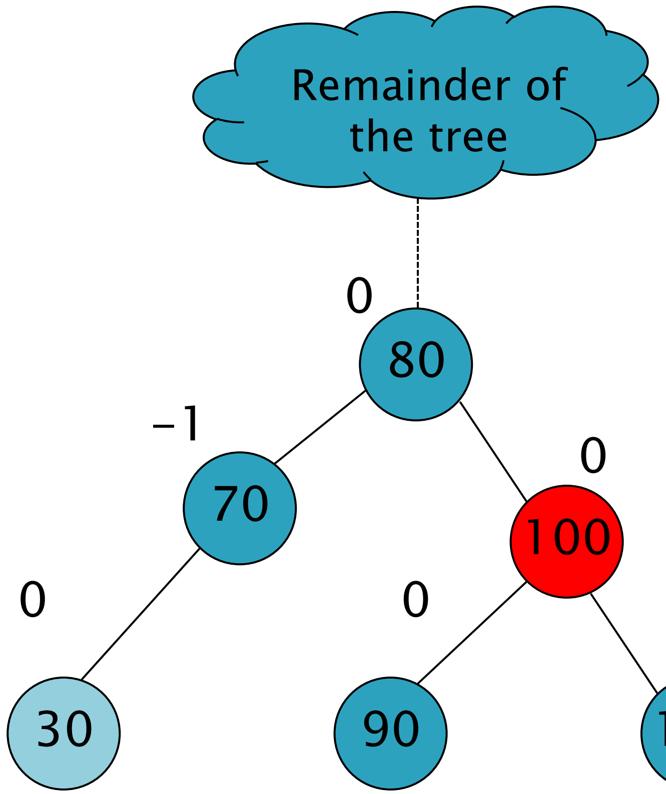
Right Rotation (Single)
Insert 30

AVL Tree: Insert (Case 3)



Right Rotation (Single)
Insert 30

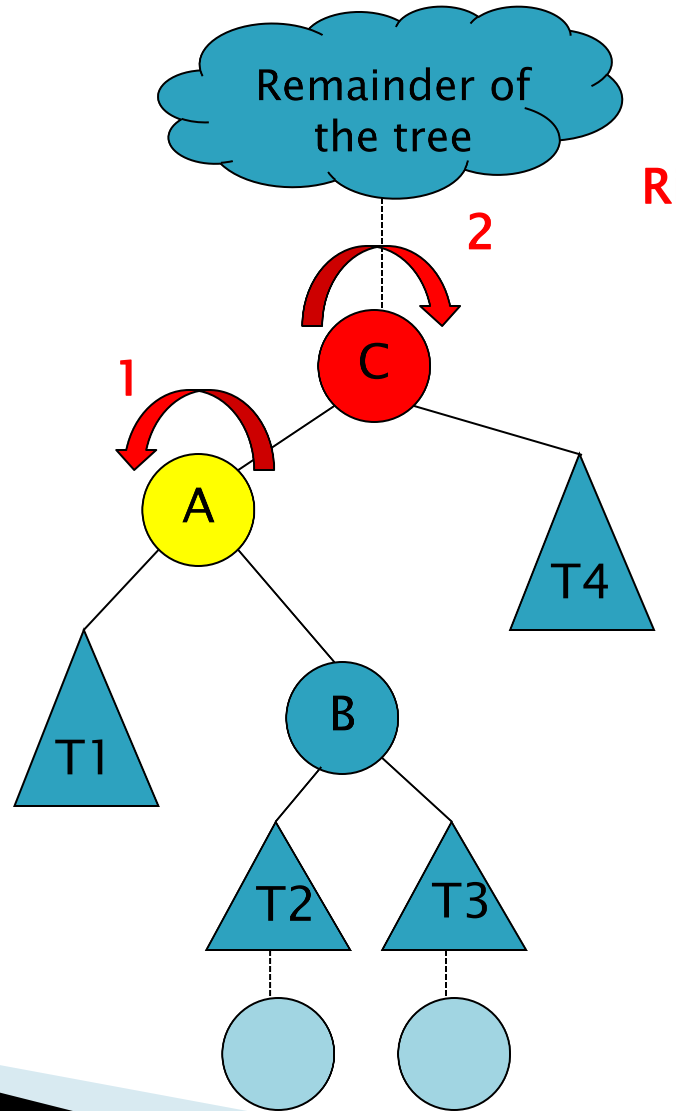
AVL Tree: Insert (Case 3)



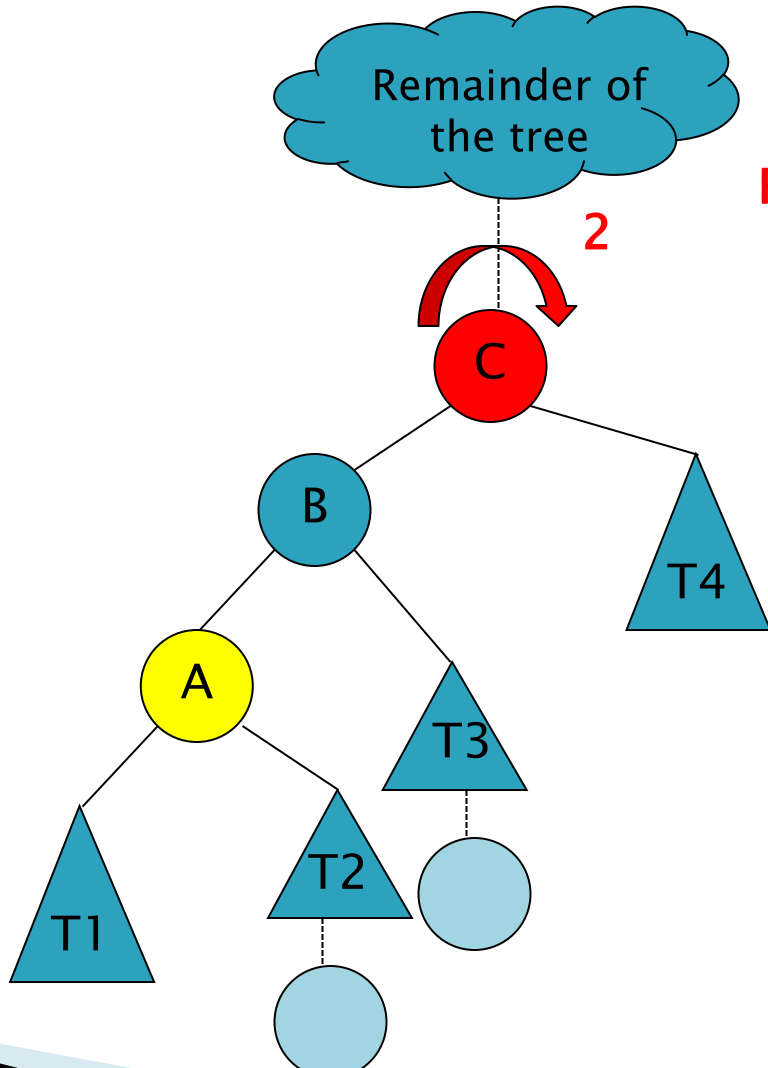
Right Rotation (Single)
Insert 30

AVL Tree: Insert (Case 3)

Right-Left Rotation (Double)



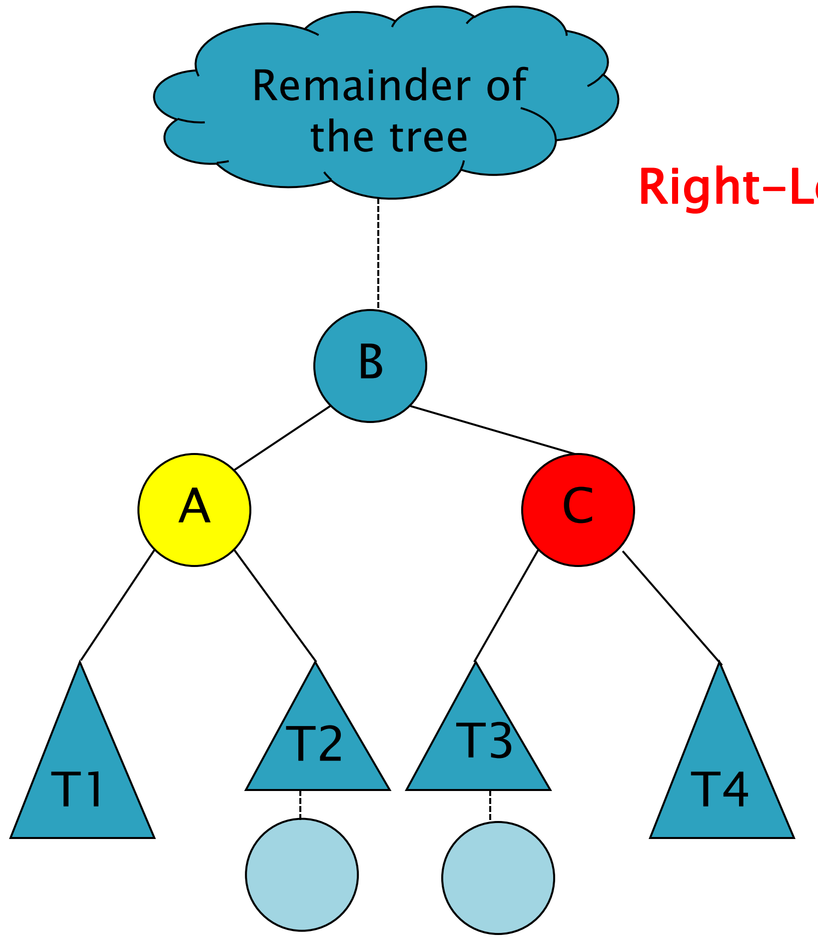
AVL Tree: Insert (Case 3)



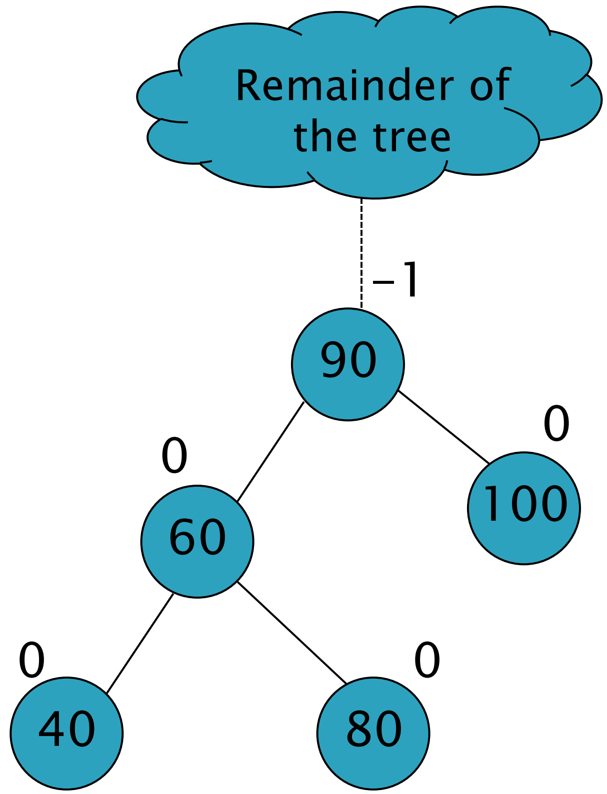
Right-Left Rotation (Double)

AVL Tree: Insert (Case 3)

Right-Left Rotation (Double)

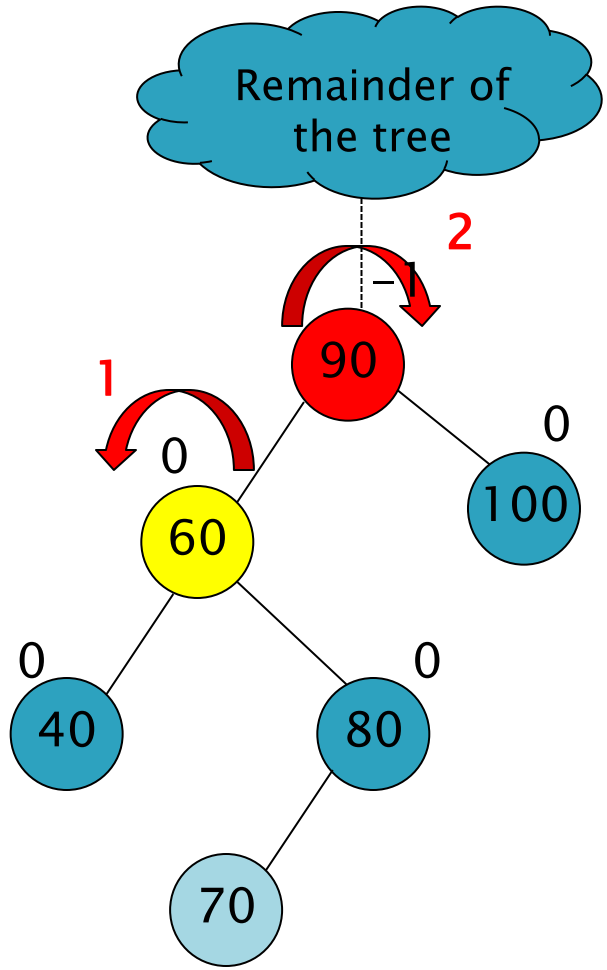


AVL Tree: Insert (Case 3)



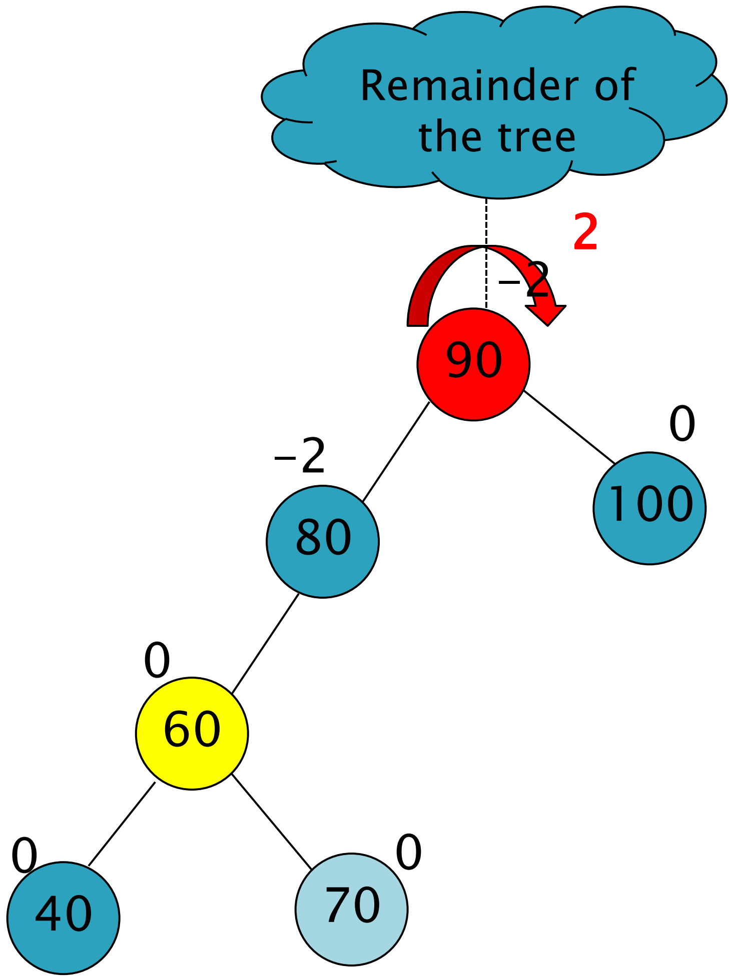
Right-Left Rotation (Double)
Insert 70

AVL Tree: Insert (Case 3)



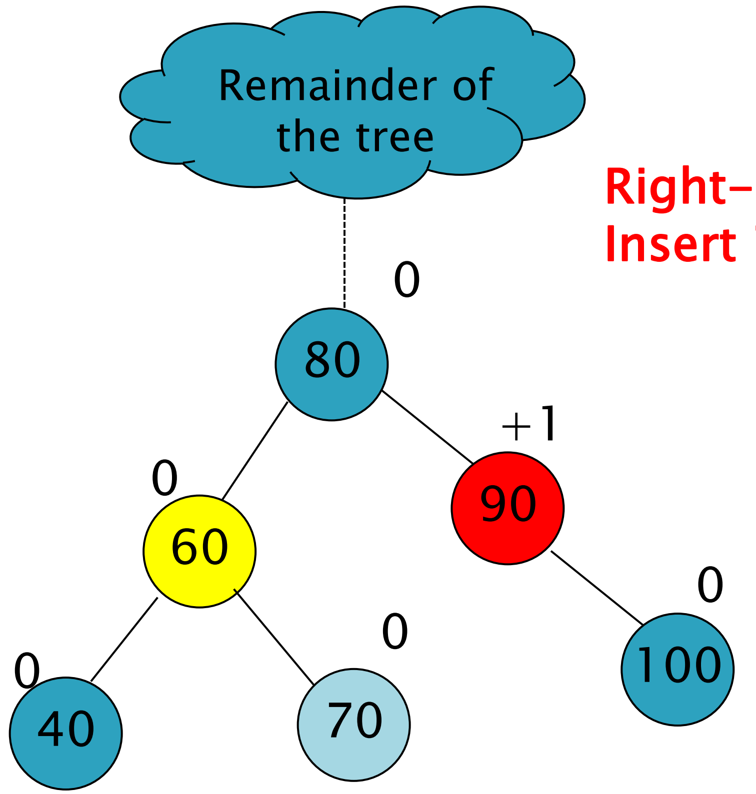
Right-Left Rotation (Double)
Insert 70

AVL Tree: Insert (Case 3)



Right-Left Rotation (Double)
Insert 70

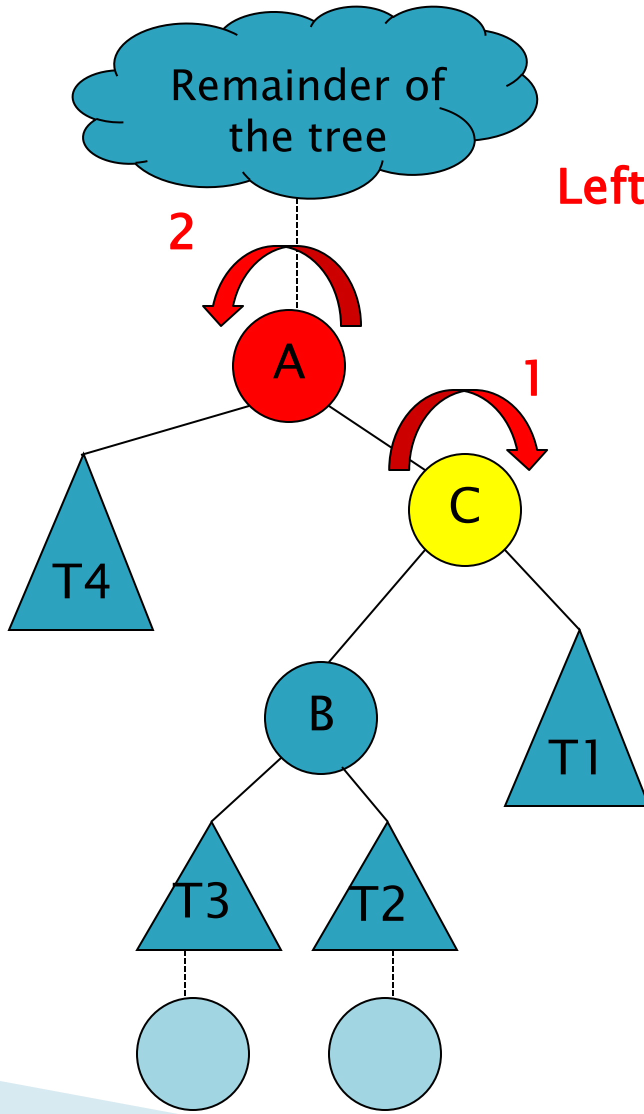
AVL Tree: Insert (Case 3)



Right-Left Rotation (Double)
Insert 70

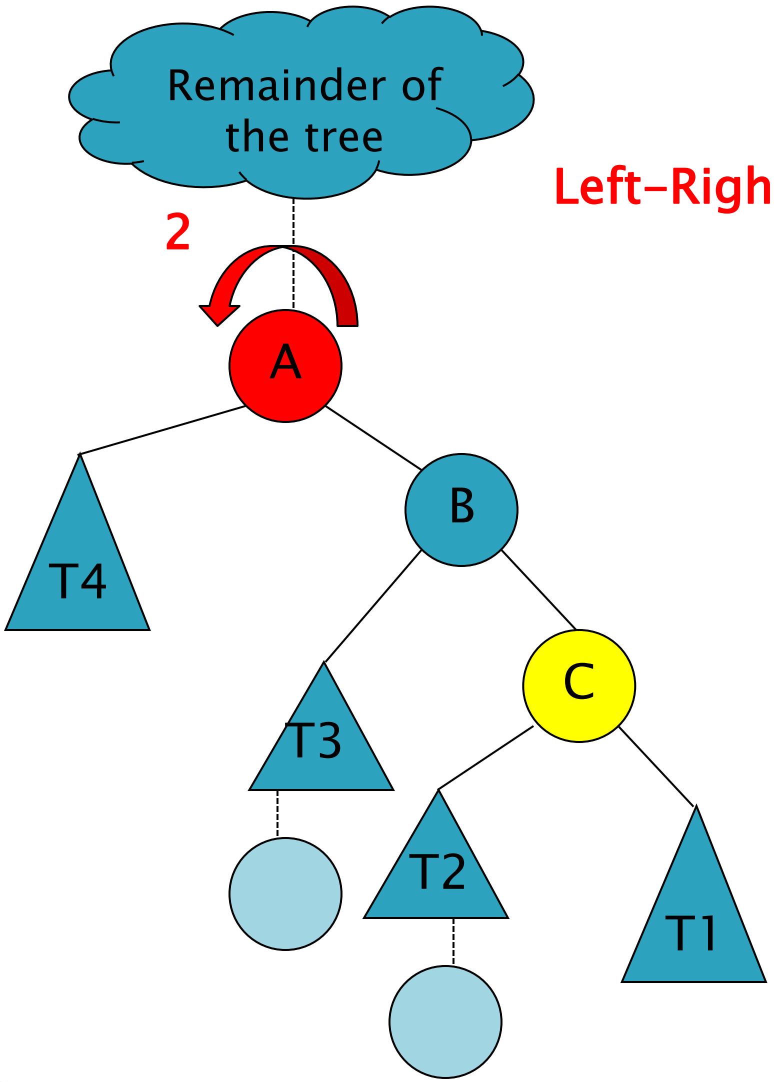
AVL Tree: Insert (Case 3)

Left-Right Rotation (Double)



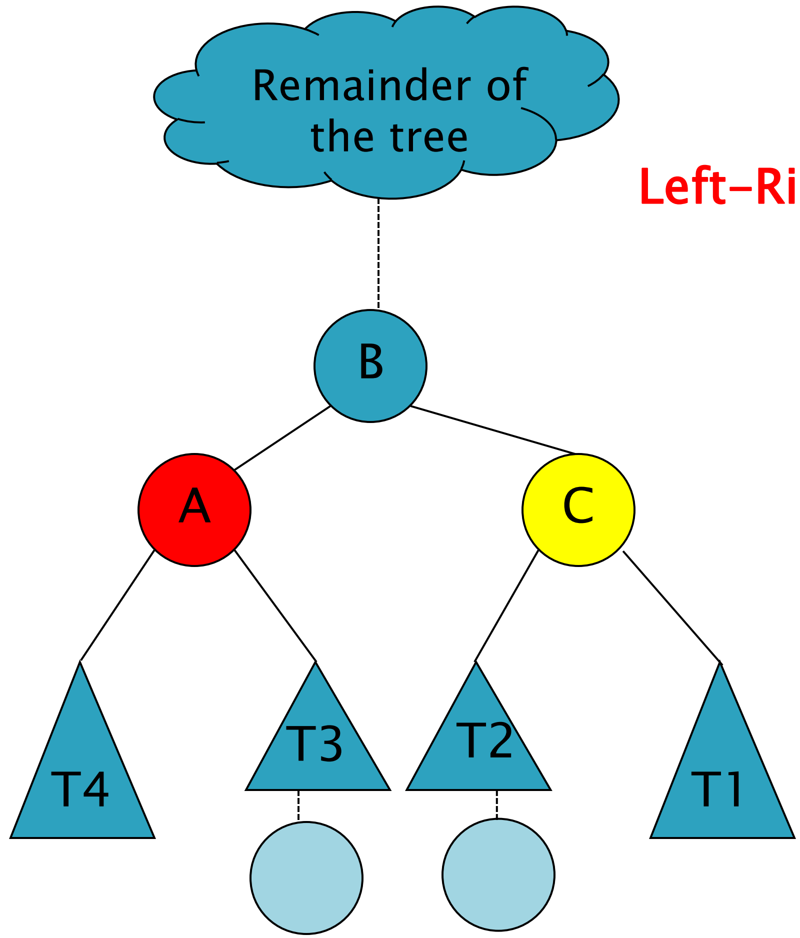
AVL Tree: Insert (Case 3)

Left-Right Rotation (Double)

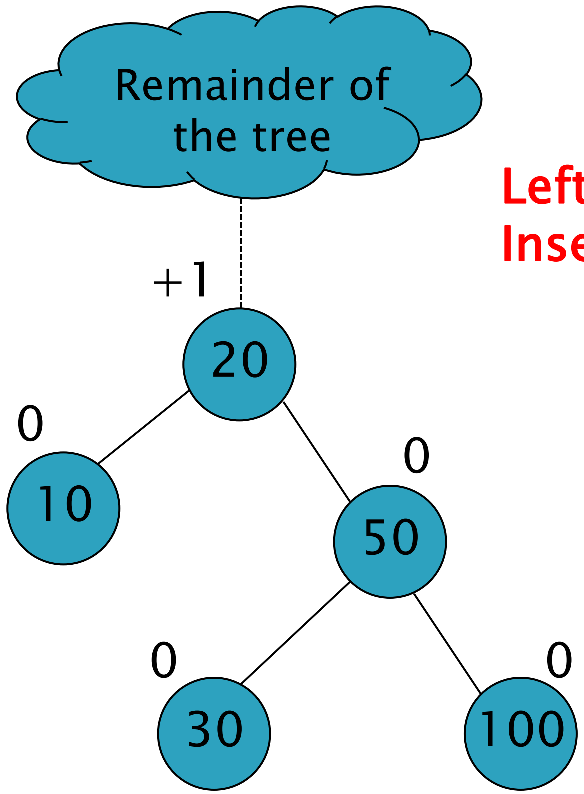


AVL Tree: Insert (Case 3)

Left-Right Rotation (Double)

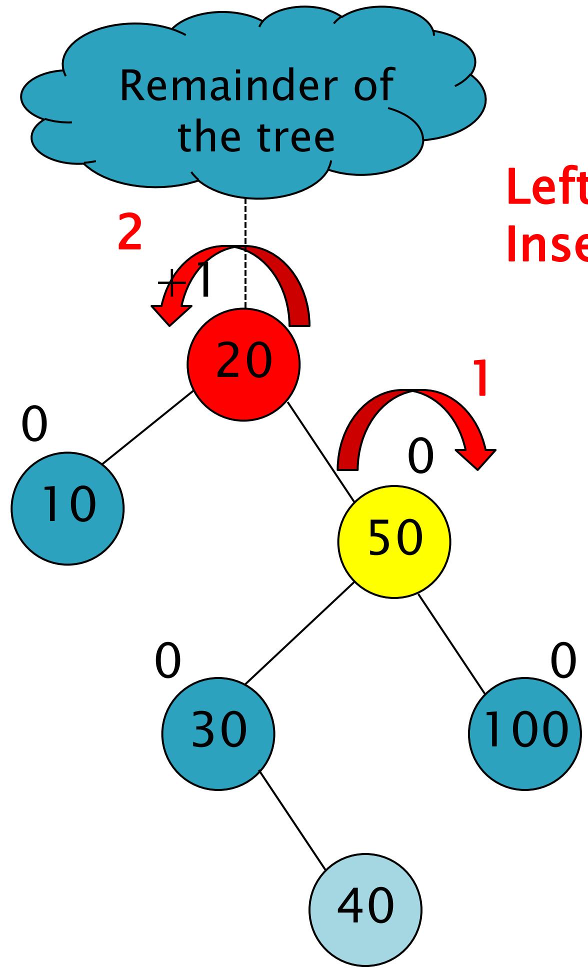


AVL Tree: Insert (Case 3)



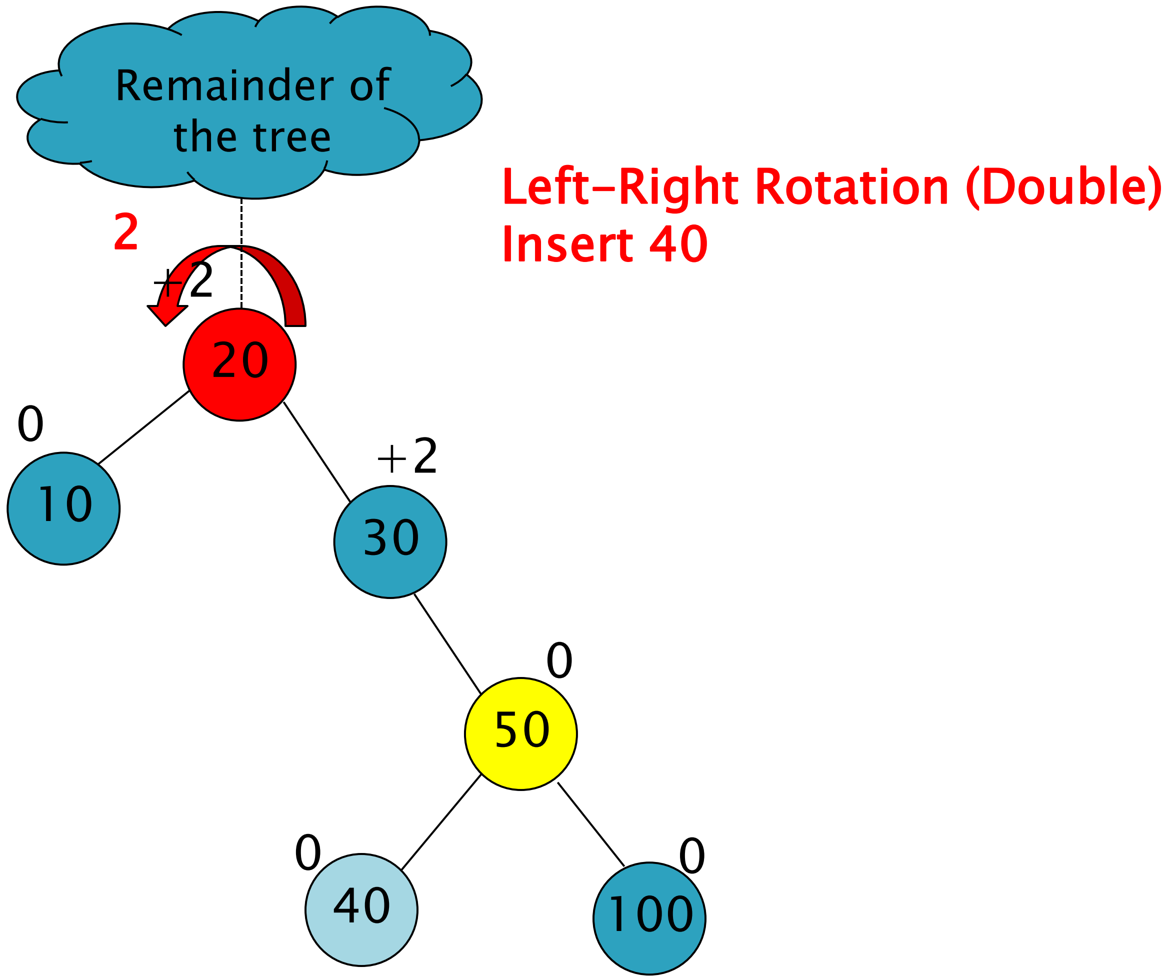
Left-Right Rotation (Double)
Insert 40

AVL Tree: Insert (Case 3)

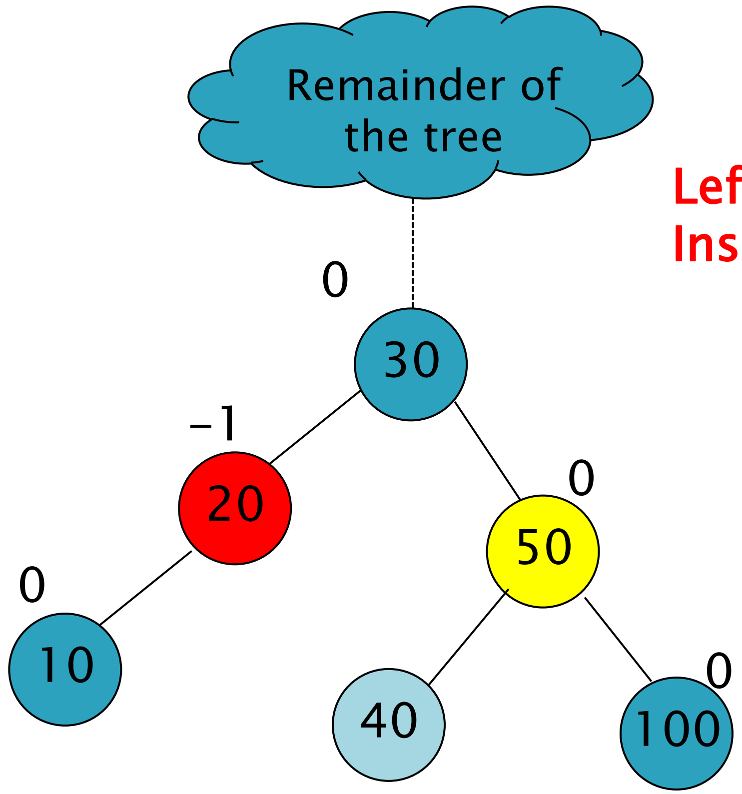


Left-Right Rotation (Double)
Insert 40

AVL Tree: Insert (Case 3)



AVL Tree: Insert (Case 3)



Left-Right Rotation (Double)
Insert 40

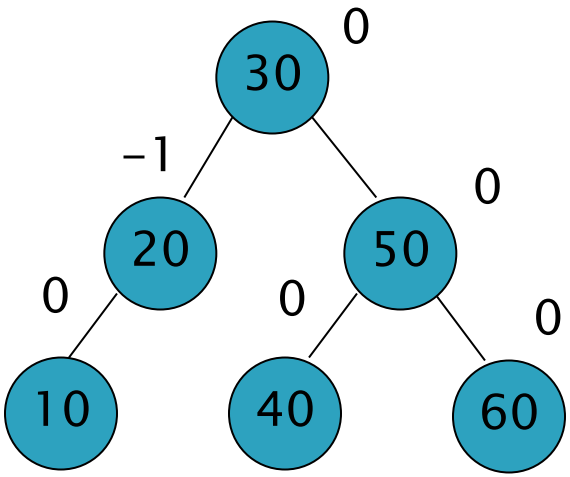
AVL Tree: Delete

- <u>Step 1:</u>
  Delete the node as in BSTs. Remember there are three cases for BST deletion.
- <u>Step 2:</u>
  For <u>each node</u> on the path from the root to deleted node, check if the node has become imbalanced; if yes perform rotation operations otherwise update balance factors and exit. Three cases can arise for each node p, in the path.

AVL Tree: Delete

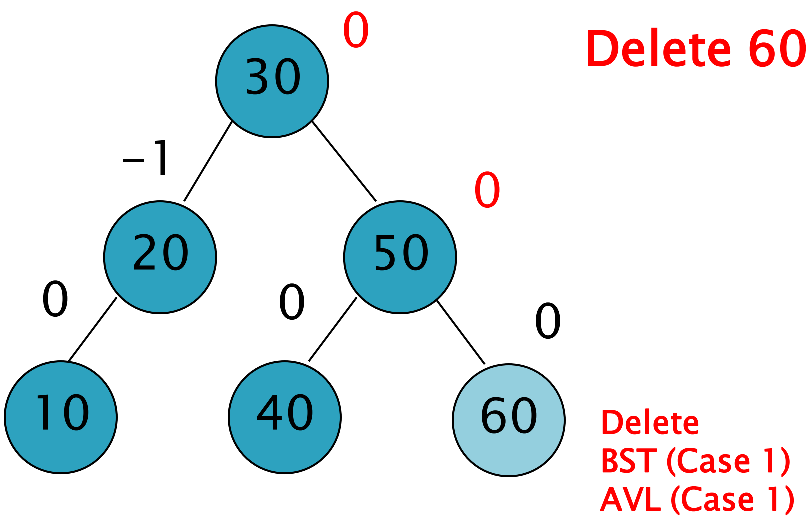
- **Case 1:**
  Node p has balance factor 0. No adjustment required.
- **Case 2:**
  Node p has balance factor of +1 or -1 and a node was deleted from the taller sub-trees. No adjustment required.
- **Case 3:**
  Node p has balance factor of +1 or -1 and a node was deleted from the shorter sub-trees. **Adjustment required.**

AVL Tree: Delete (Case 1)



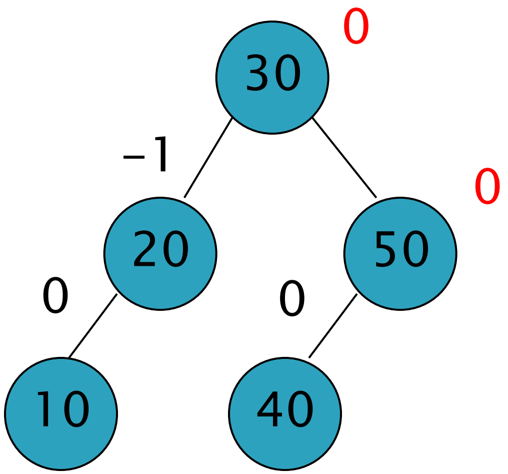
Delete 60

AVL Tree: Delete (Case 1)



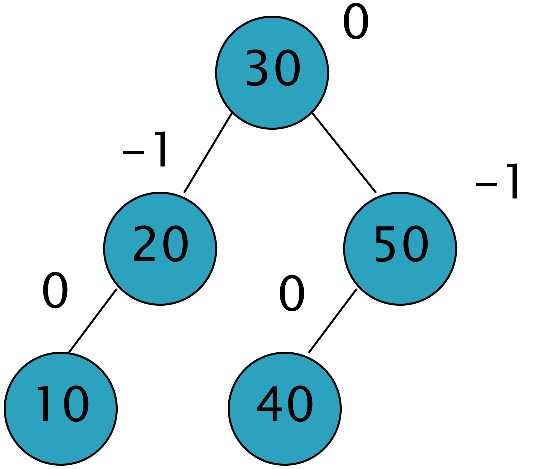
Delete 60

AVL Tree: Delete (Case 1)



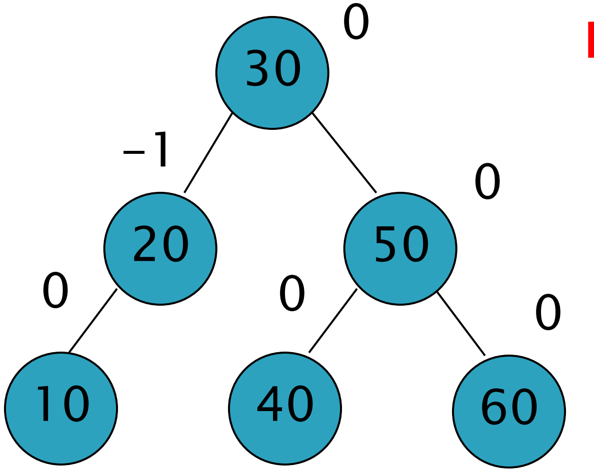
Delete 60

AVL Tree: Delete (Case 1)



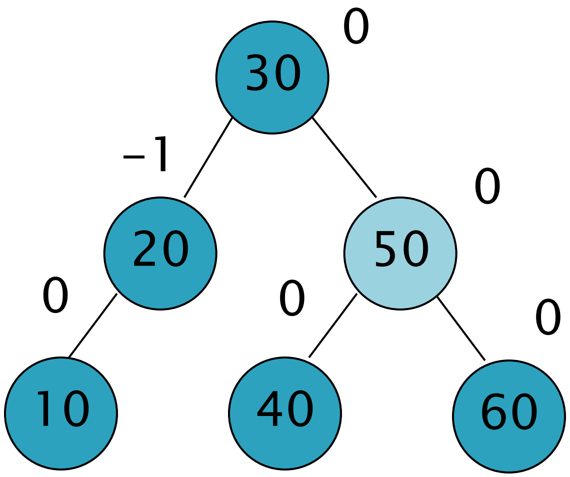
Delete 60

AVL Tree: Delete (Case 1)



Delete 50

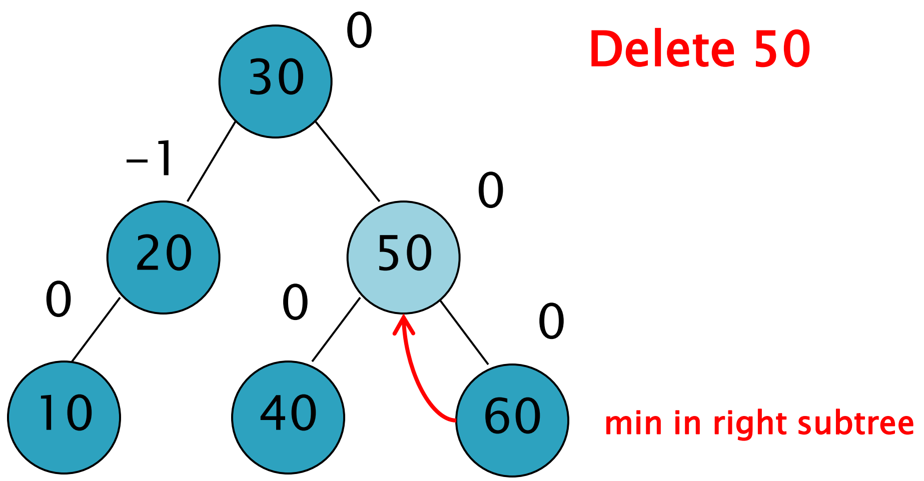
AVL Tree: Delete (Case 1)



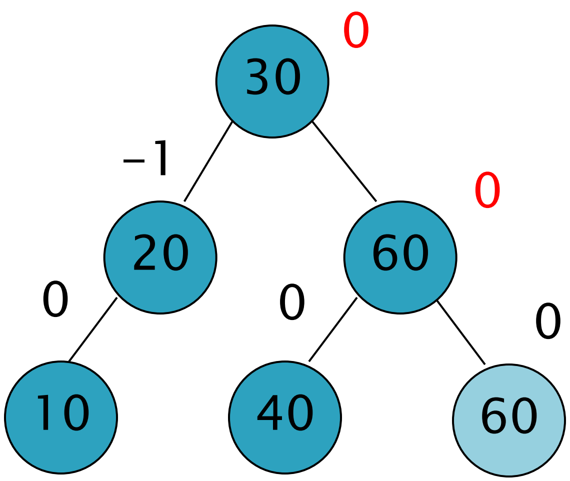
Delete 50

Delete
BST (Case 3)

AVL Tree: Delete (Case 1)



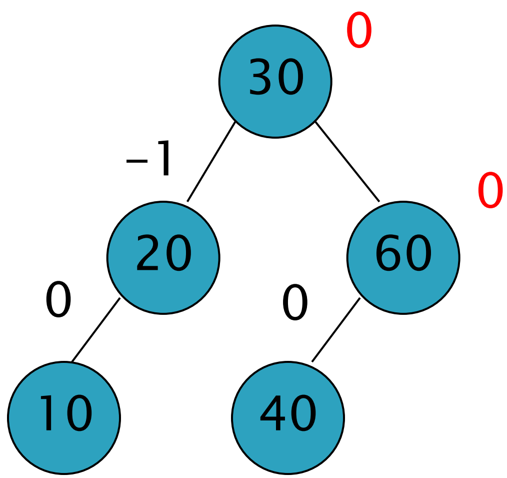
AVL Tree: Delete (Case 1)



Delete 50

Delete duplicate
BST (Case 1)

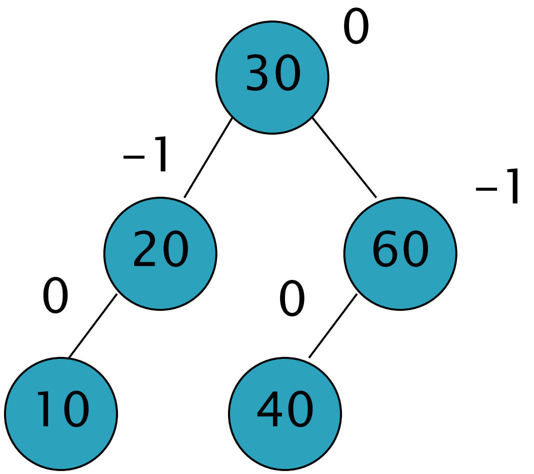
AVL Tree: Delete (Case 1)



Delete 50

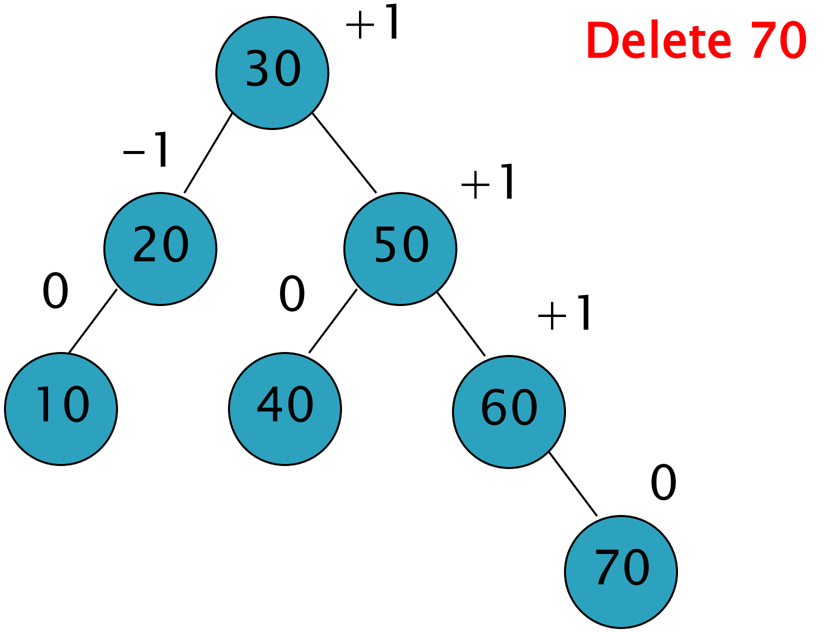
AVL (Case 1)

AVL Tree: Delete (Case 1)

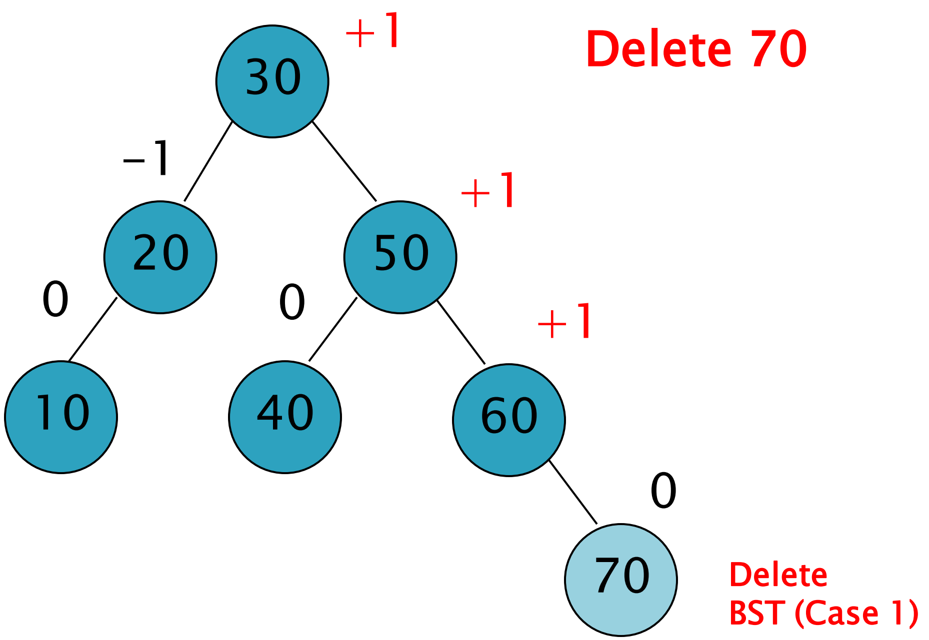


Delete 50

AVL Tree: Delete (Case 2)

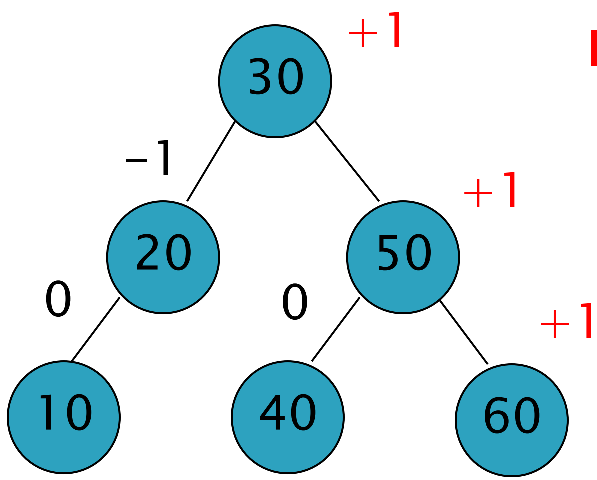


AVL Tree: Delete (Case 2)



Delete 70

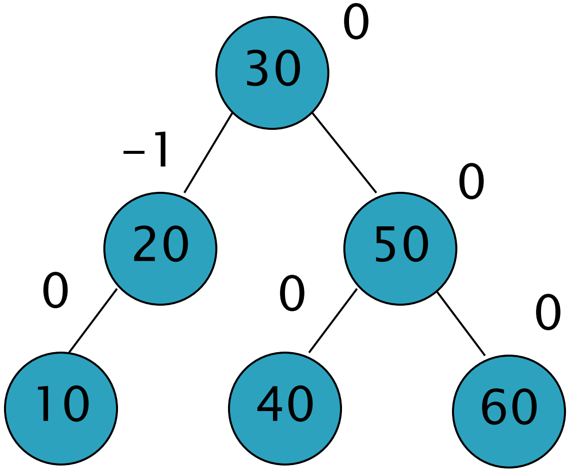
AVL Tree: Delete (Case 2)



Delete 70

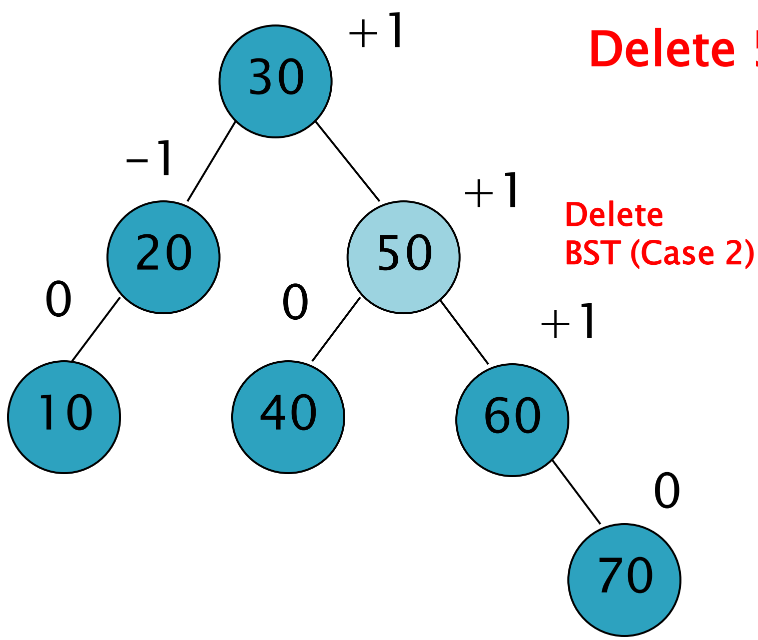
AVL (Case 2)

AVL Tree: Delete (Case 2)



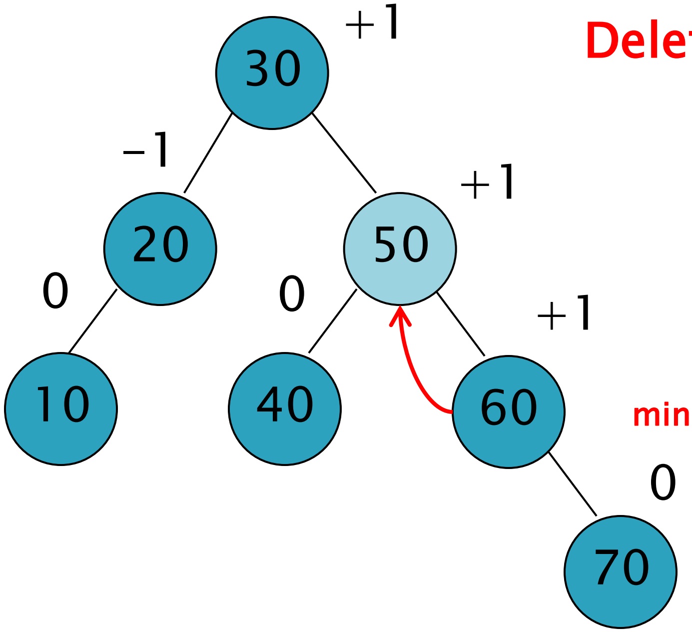
Delete 70

AVL Tree: Delete (Case 2)



Delete 50

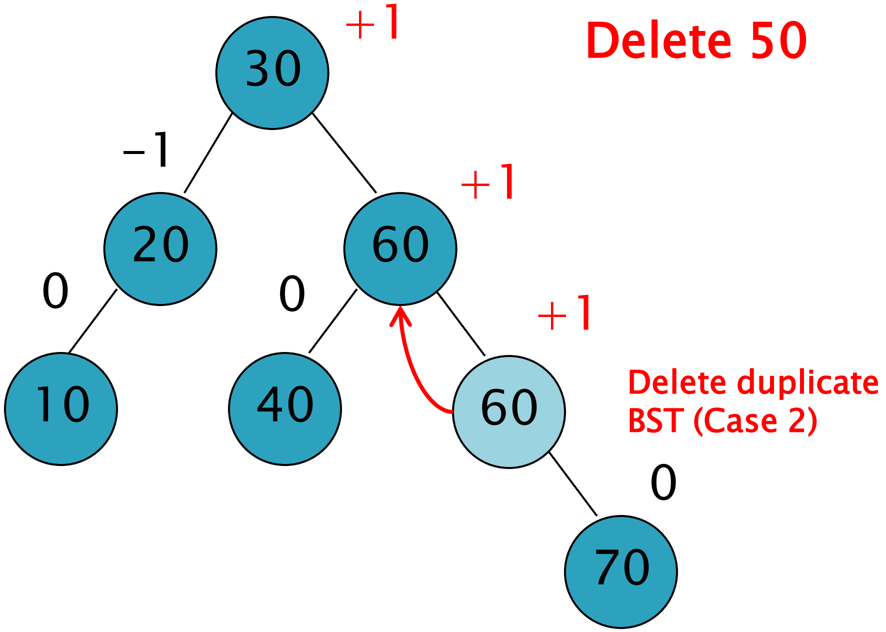
AVL Tree: Delete (Case 2)



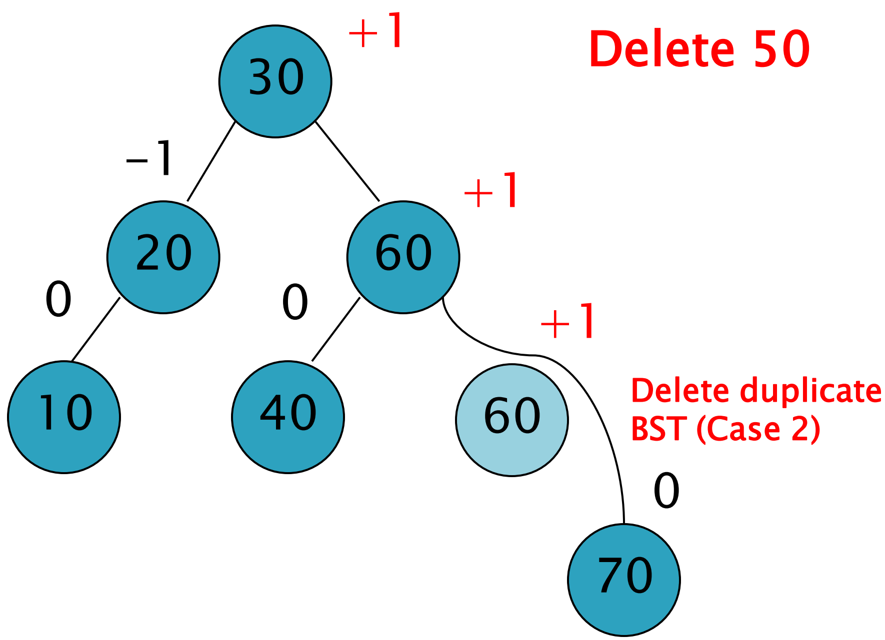
Delete 50

min in right subtree

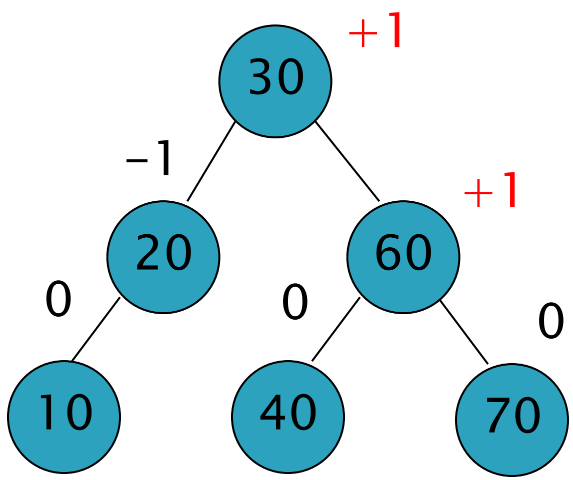
AVL Tree: Delete (Case 2)



AVL Tree: Delete (Case 2)



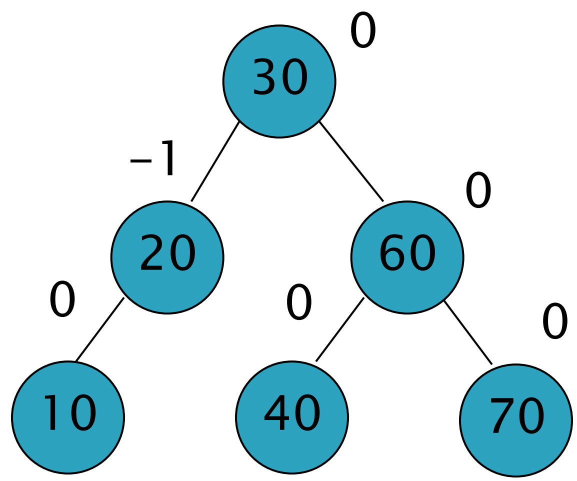
AVL Tree: Delete (Case 2)



Delete 50

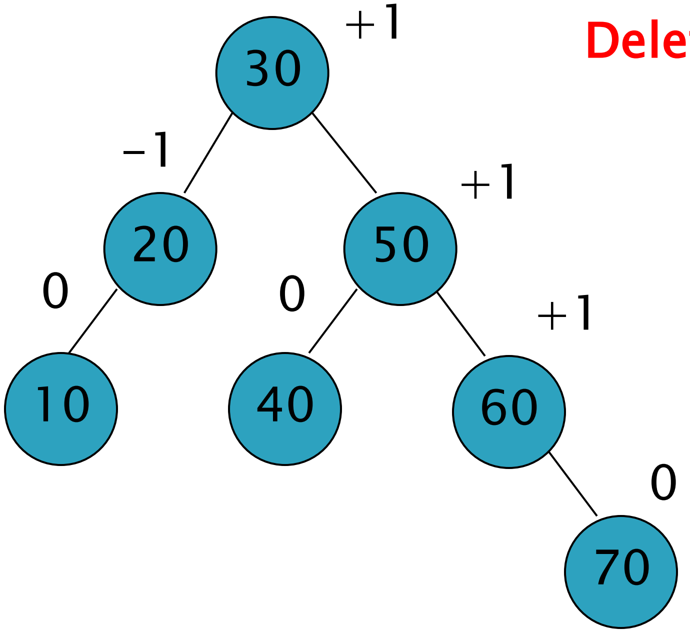
AVL (Case 2)

AVL Tree: Delete (Case 2)



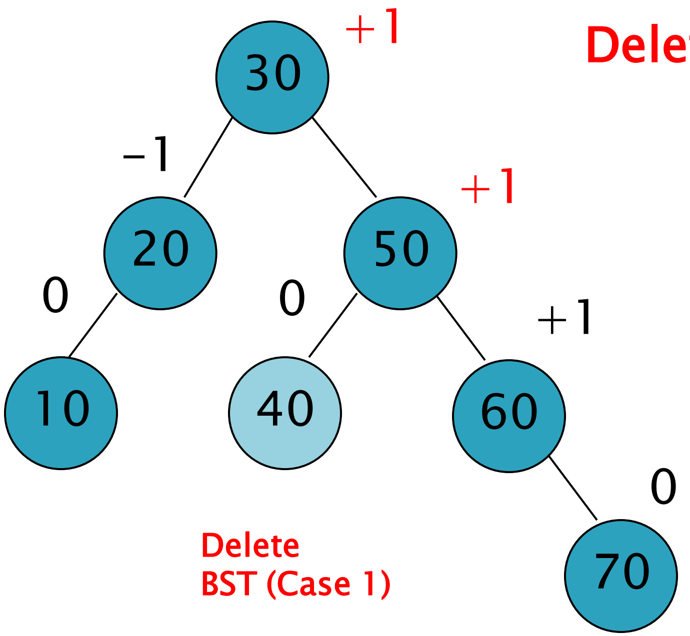
Delete 50

AVL Tree: Delete (Case 3)



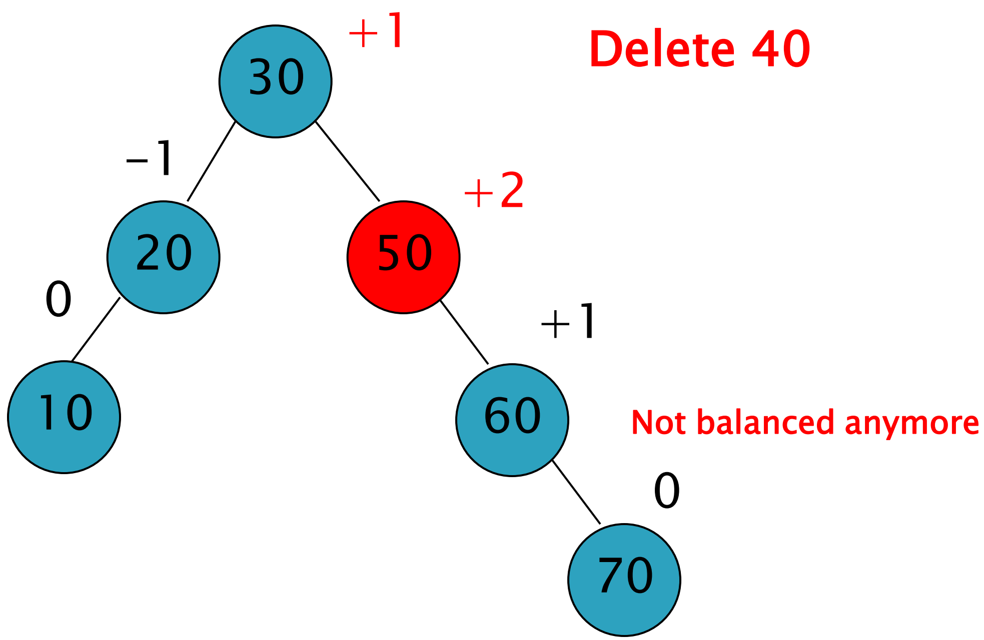
Delete 40

AVL Tree: Delete (Case 3)



Delete 40

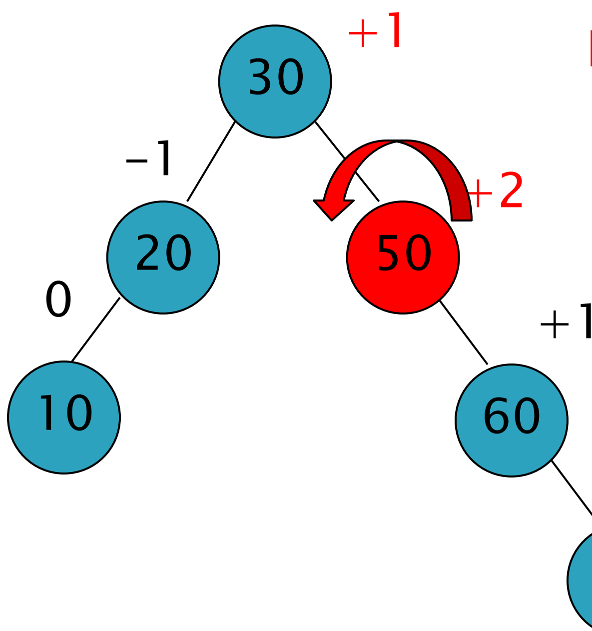
AVL Tree: Delete (Case 3)



AVL Tree: Delete (Case 3)

- Like insertion, when the tree become unbalanced after deletion, rotation need to be done.
- Like before, there are four cases:
  - Left Rotation (Single)
  - Right Rotation (Single)
  - Left–Right Rotations (Double)
  - Right–Left Rotations (Double)
- Rotation need to be done at every unbalanced nodes in the search path.

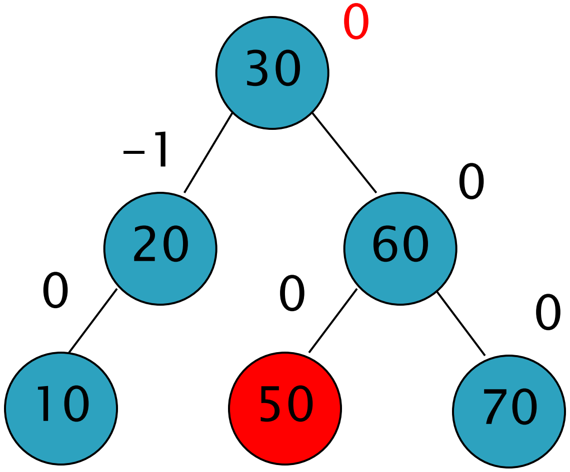
AVL Tree: Delete (Case 3)



Delete 40

Left Rotation (Single)

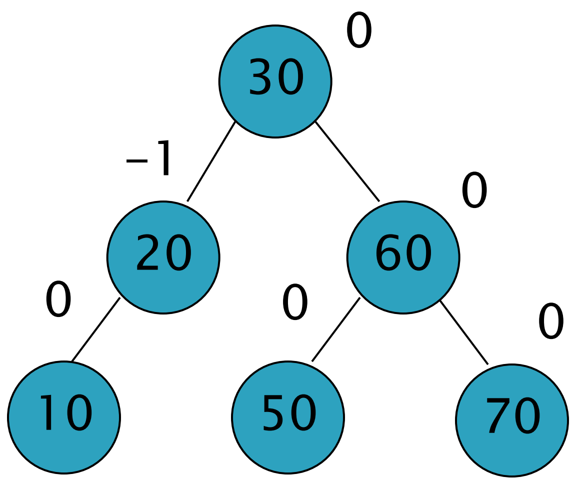
AVL Tree: Delete (Case 3)



Delete 40

No need for further rotations

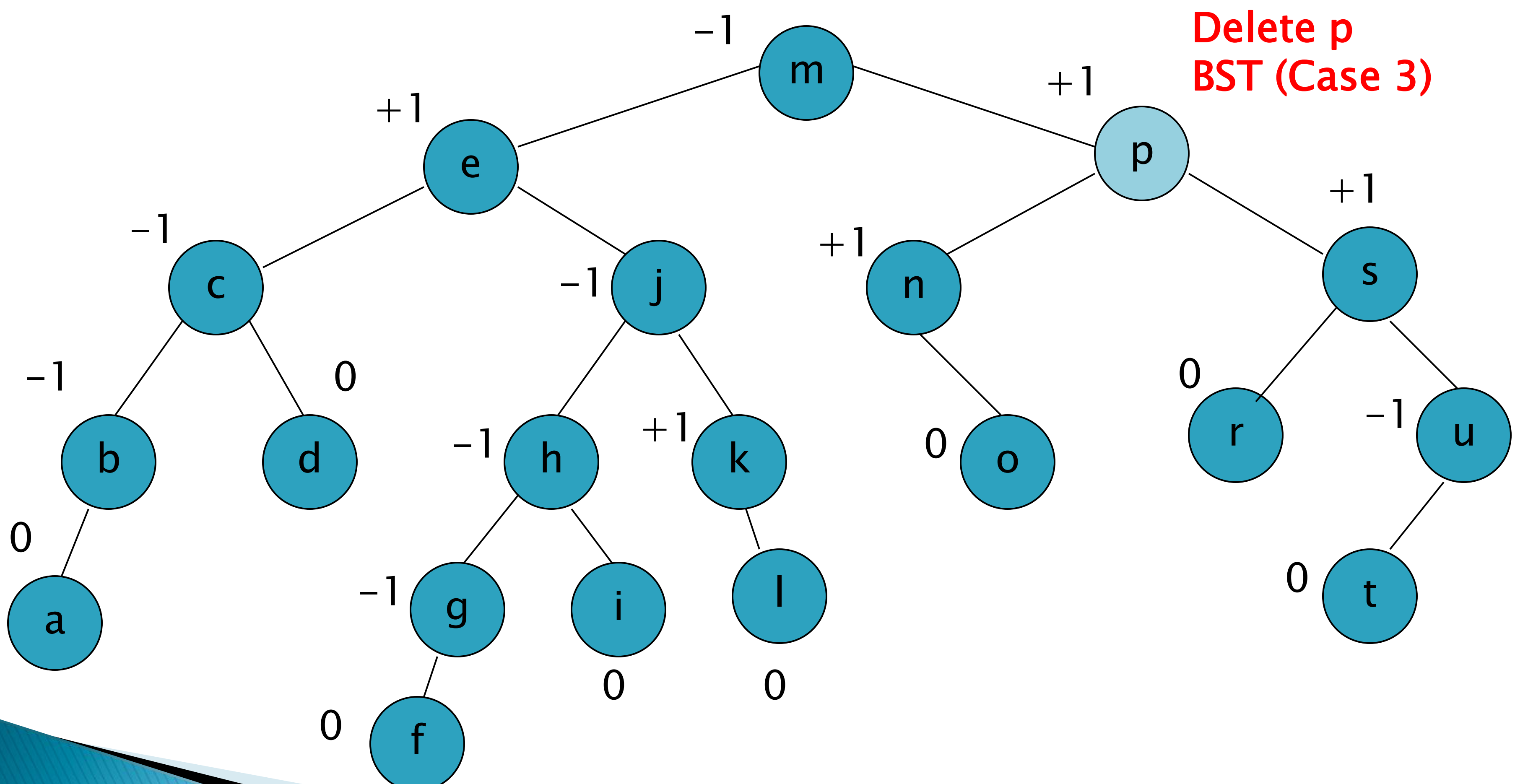
AVL Tree: Delete (Case 3)



Delete 40

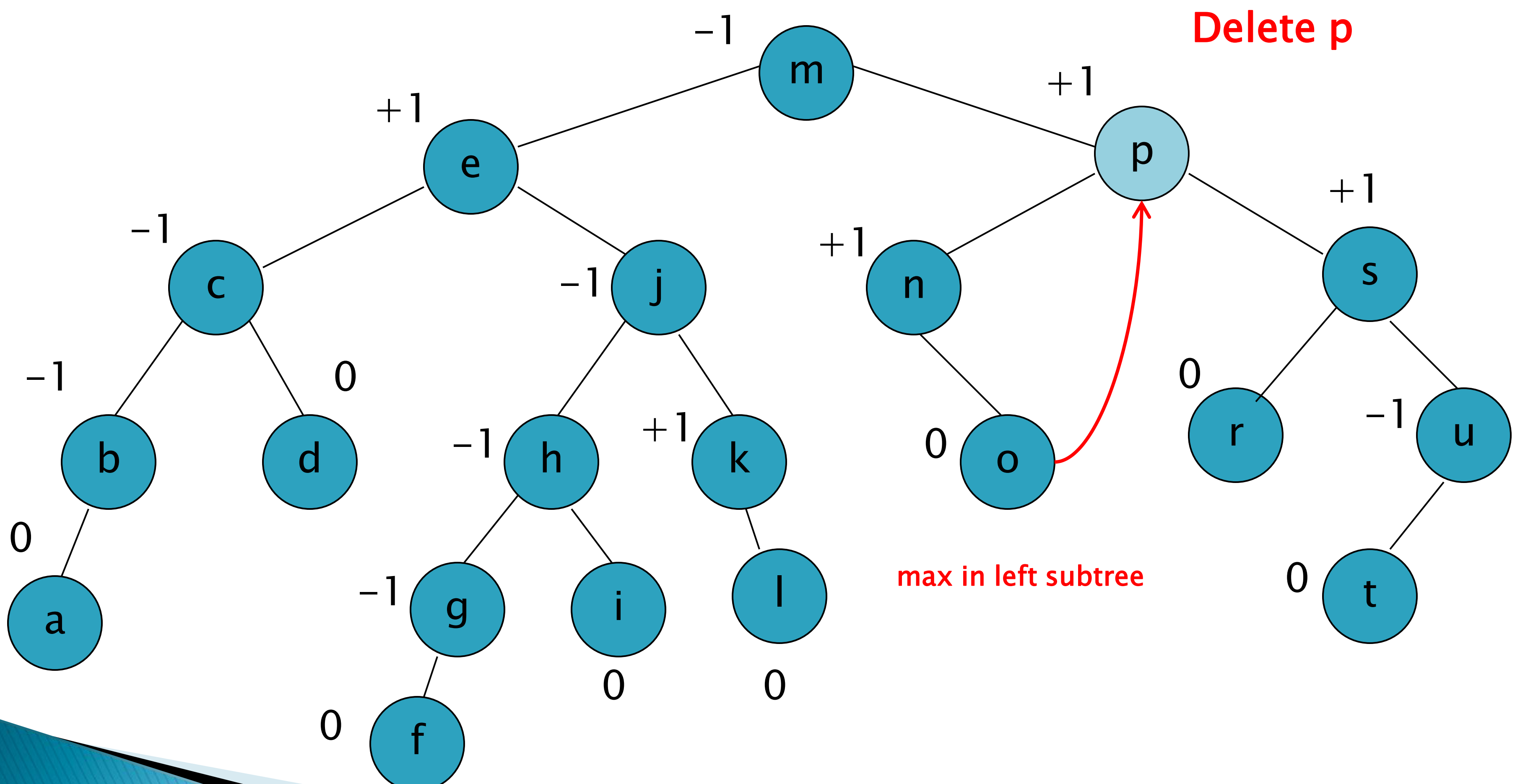
AVL Tree: Delete (Case 3)

IMPORTANT: we decided to use max in left subtree when deleting in this example (instead of min in right subtree).



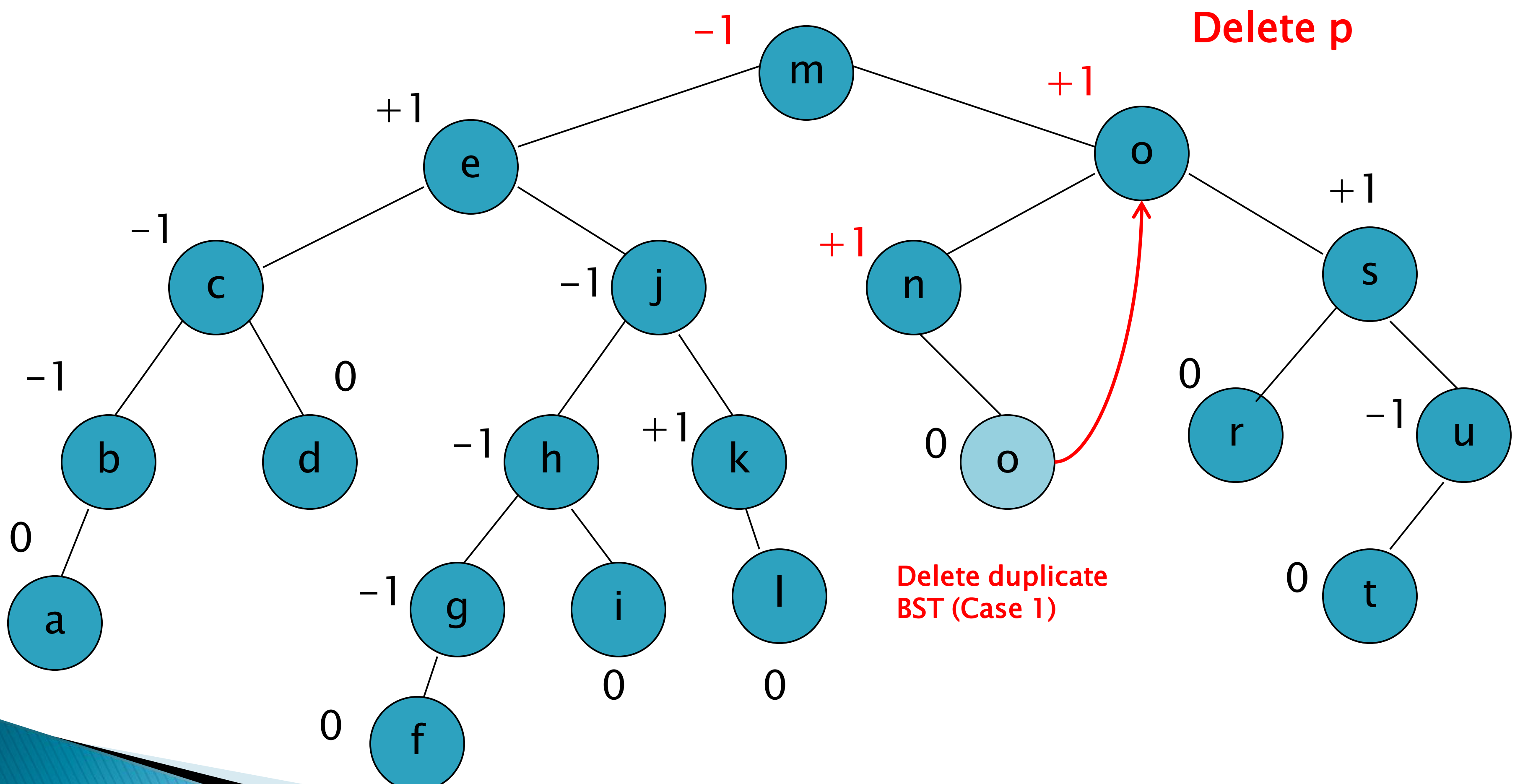
AVL Tree: Delete (Case 3)

IMPORTANT: we decided to use max in left subtree when deleting in this example (instead of min in right subtree).

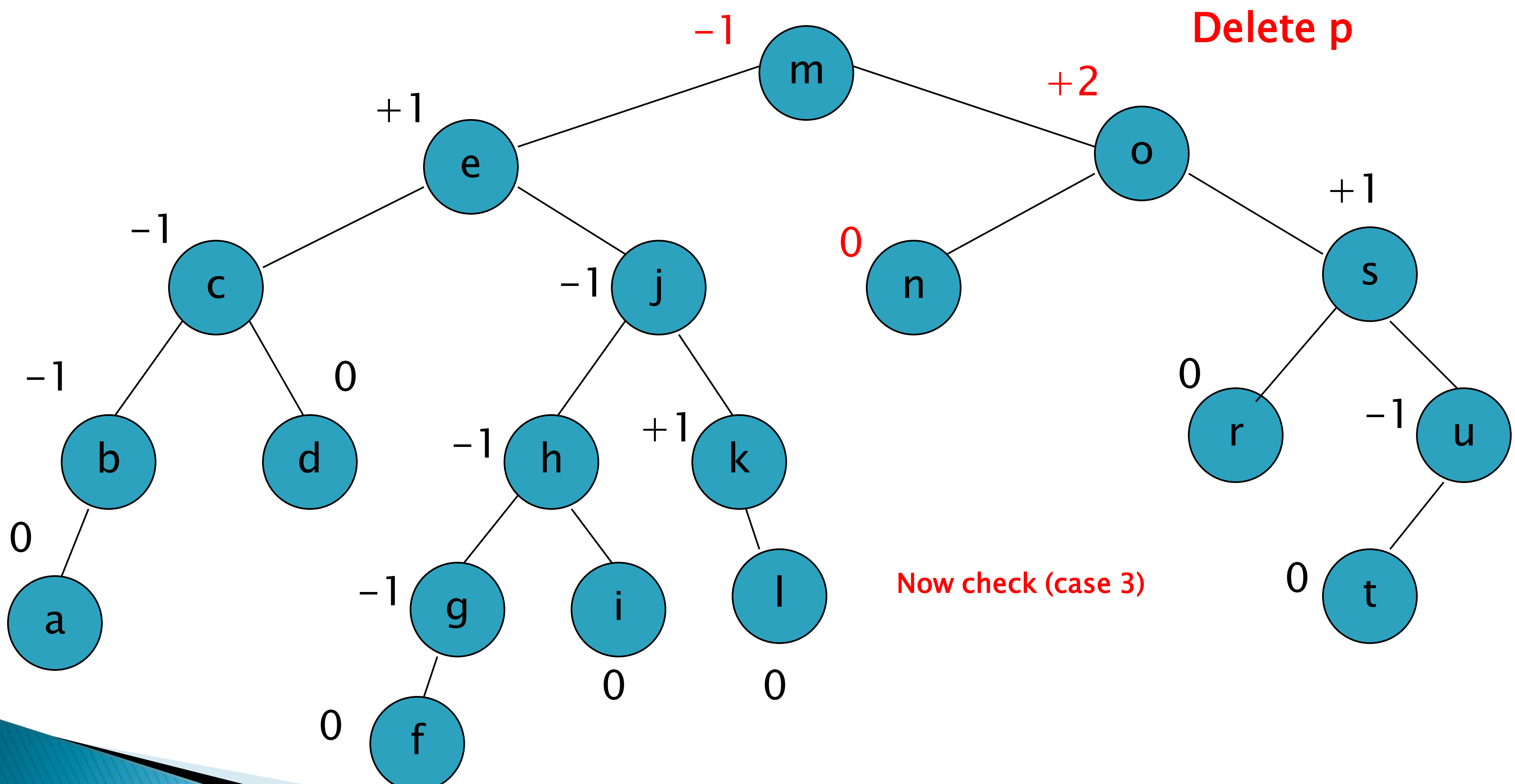


AVL Tree: Delete (Case 3)

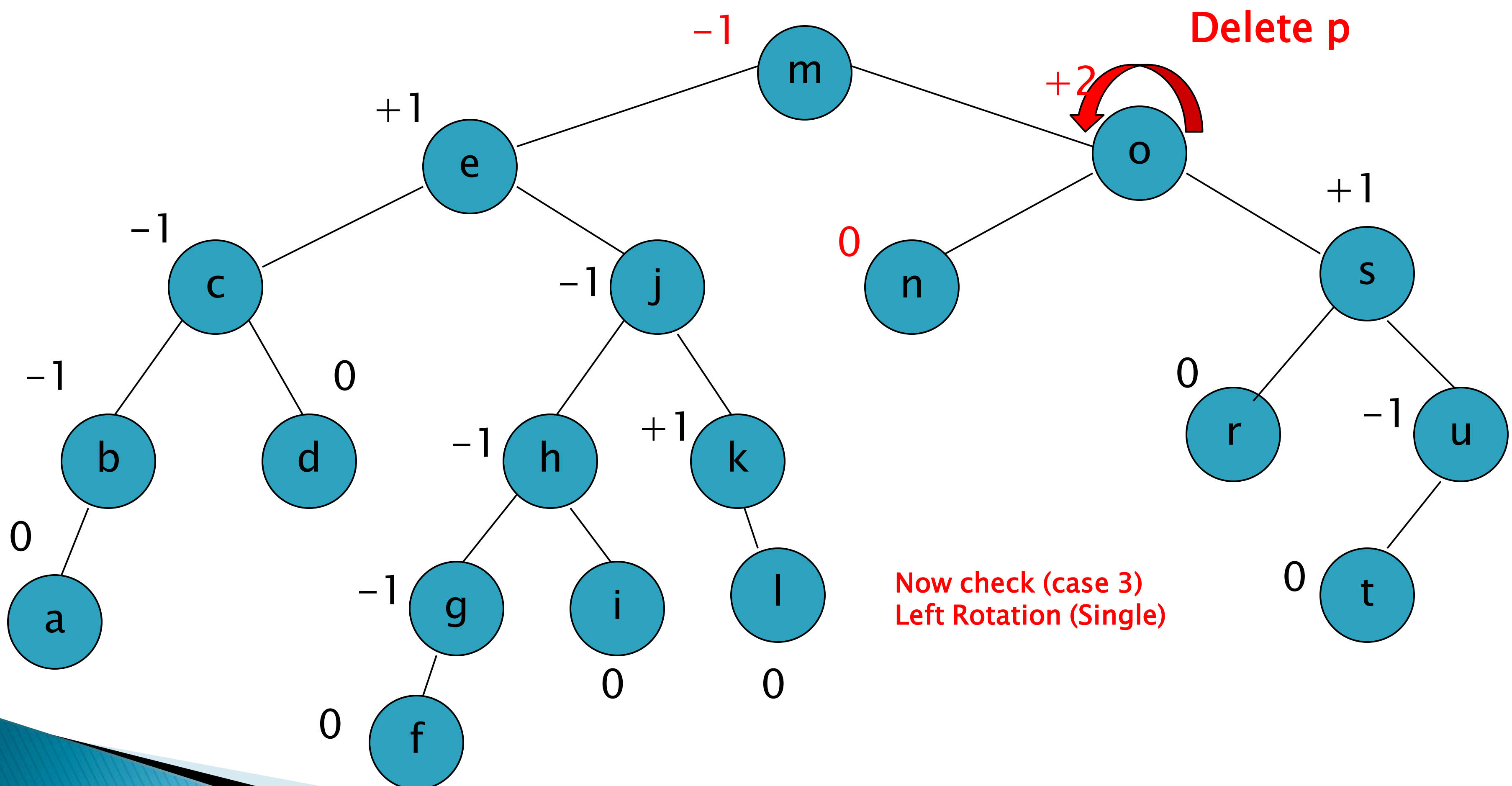
IMPORTANT: we decided to use max in left subtree when deleting in this example (instead of min in right subtree).



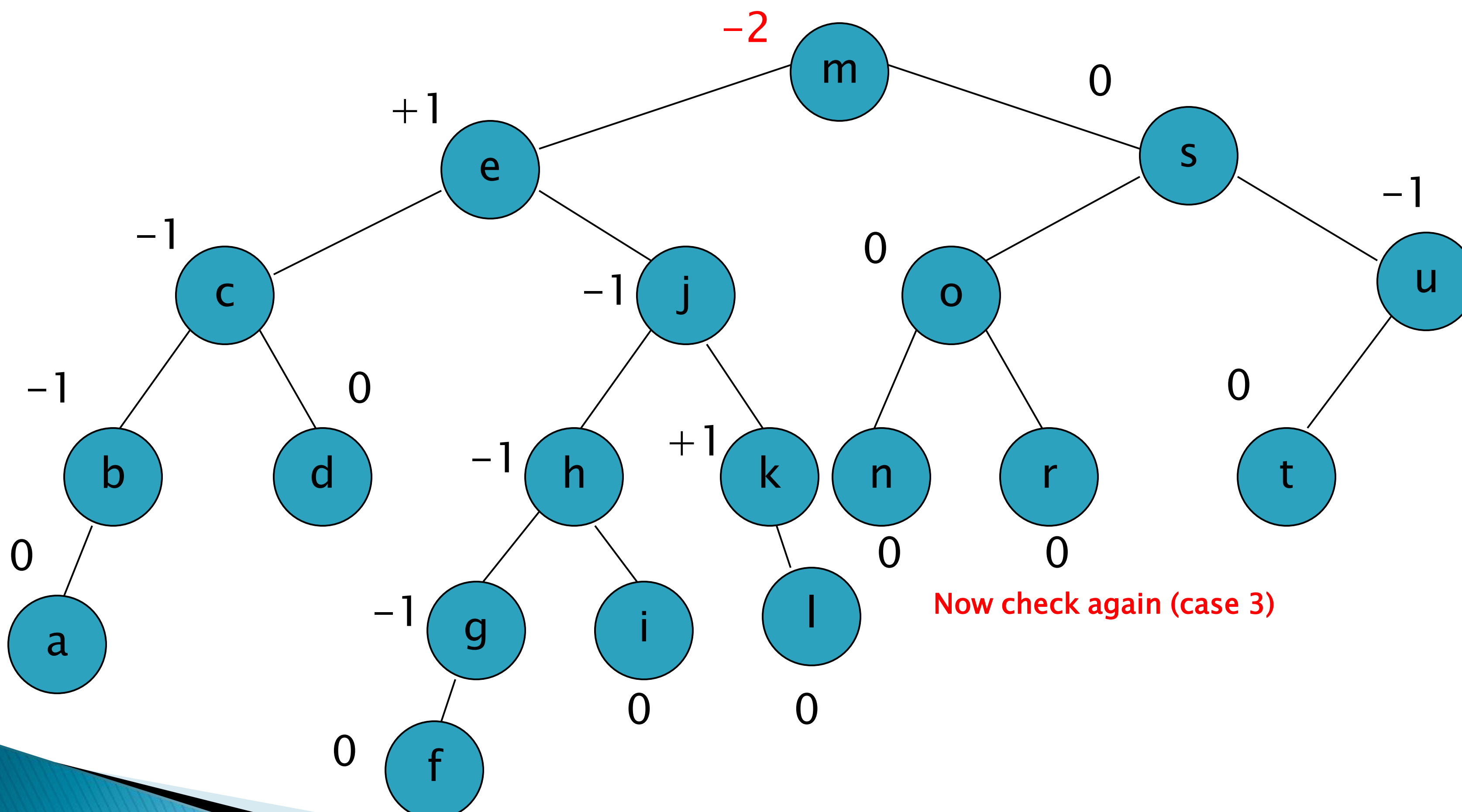
AVL Tree: Delete (Case 3)



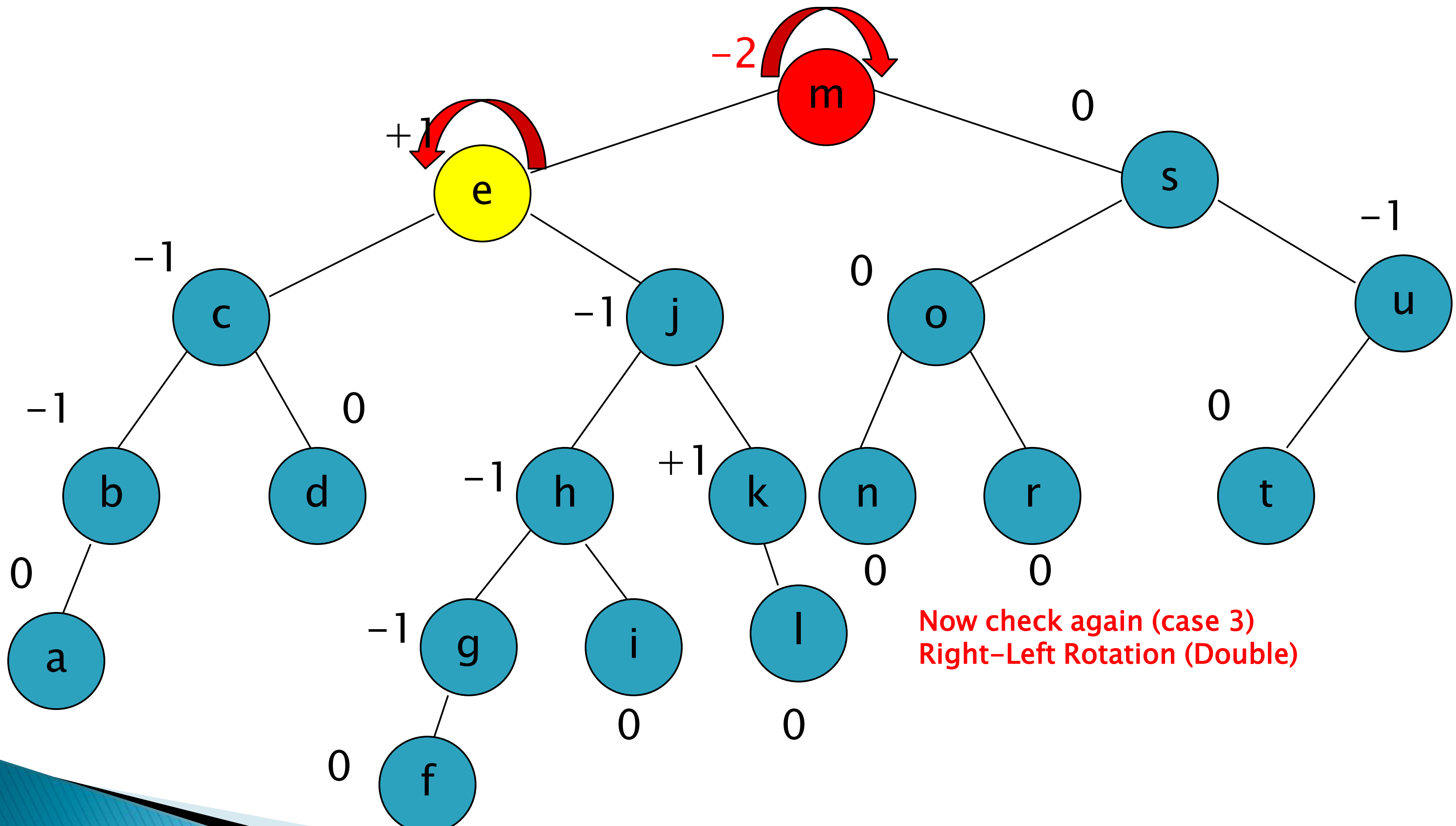
AVL Tree: Delete (Case 3)



AVL Tree: Delete (Case 3)

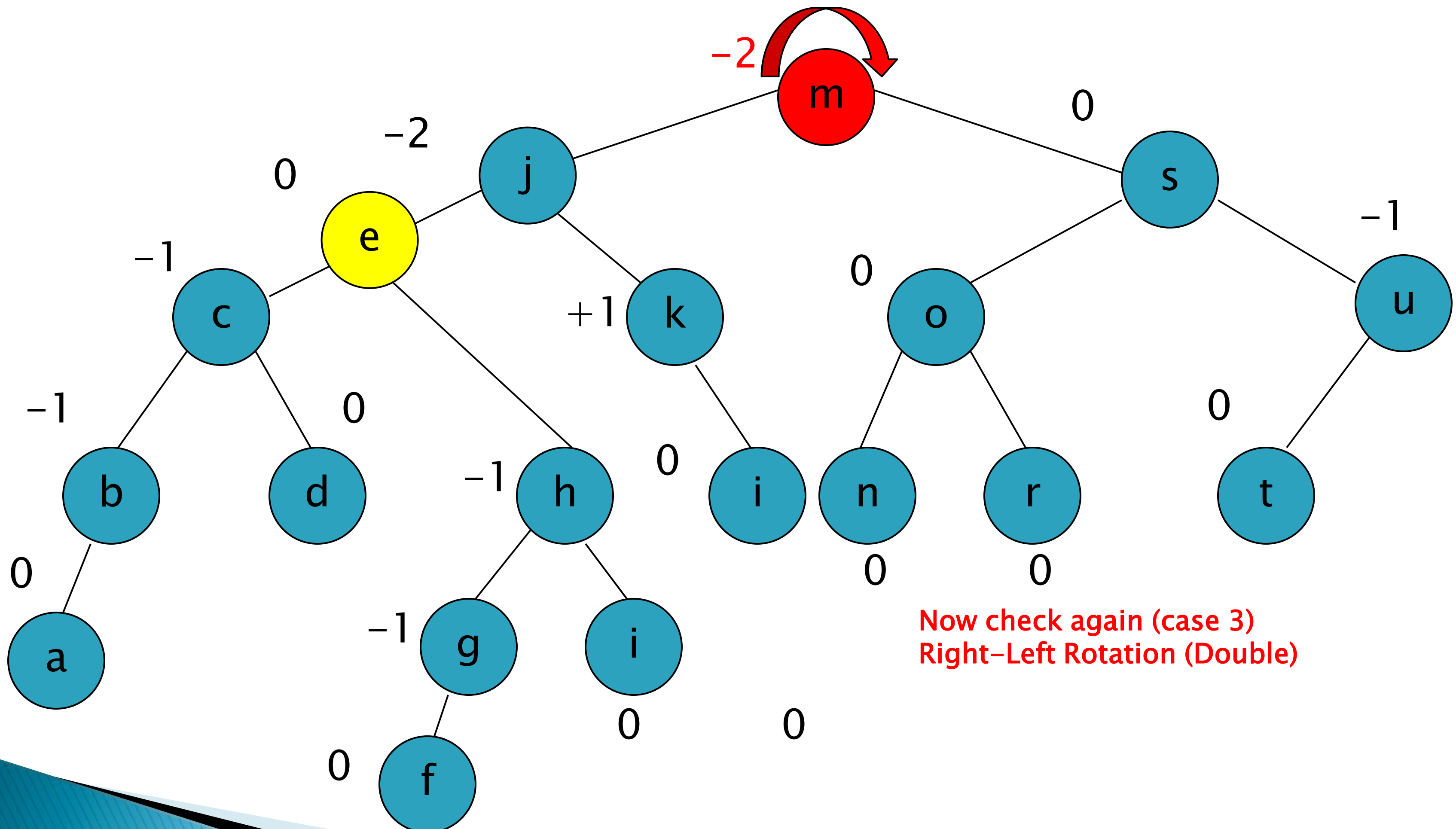


AVL Tree: Delete (Case 3)



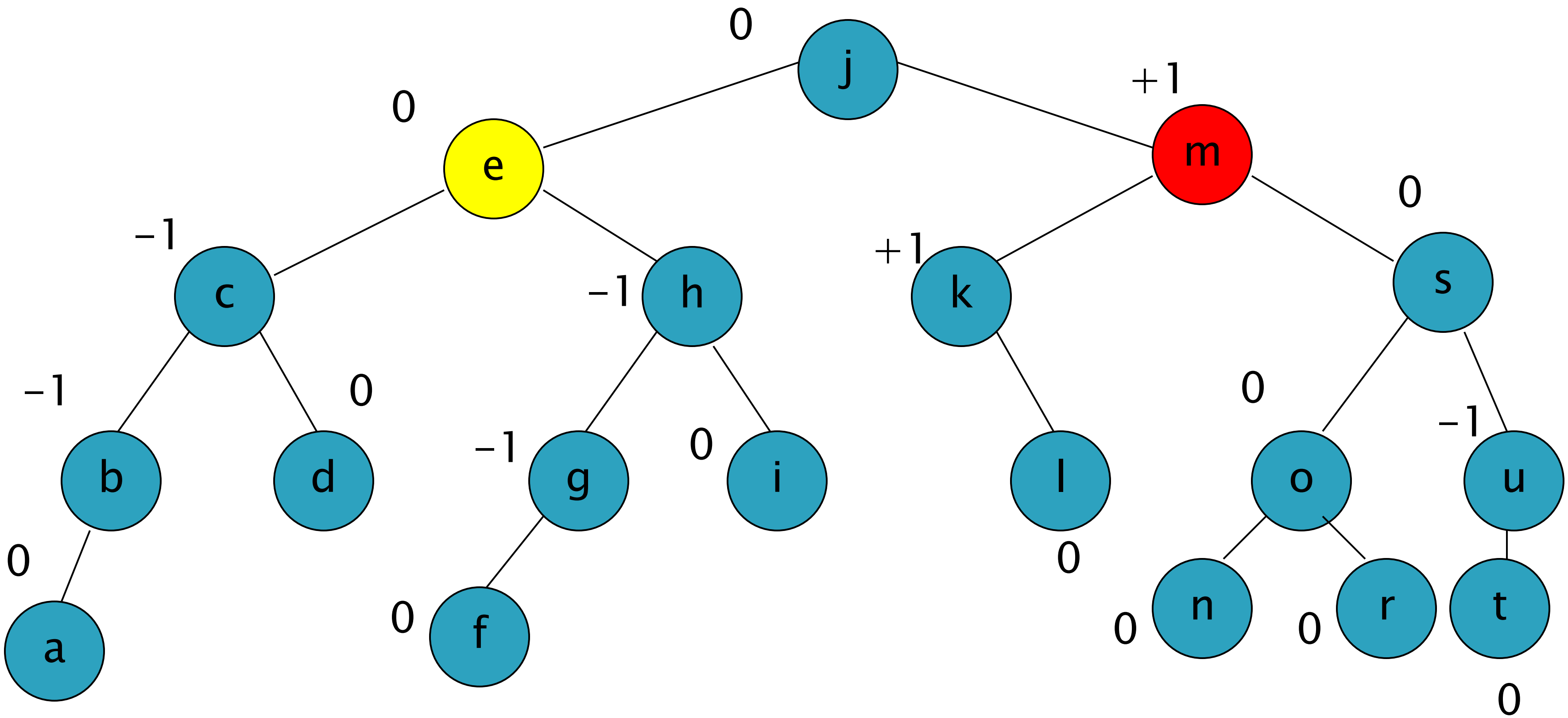
Now check again (case 3)
Right-Left Rotation (Double)

AVL Tree: Delete (Case 3)

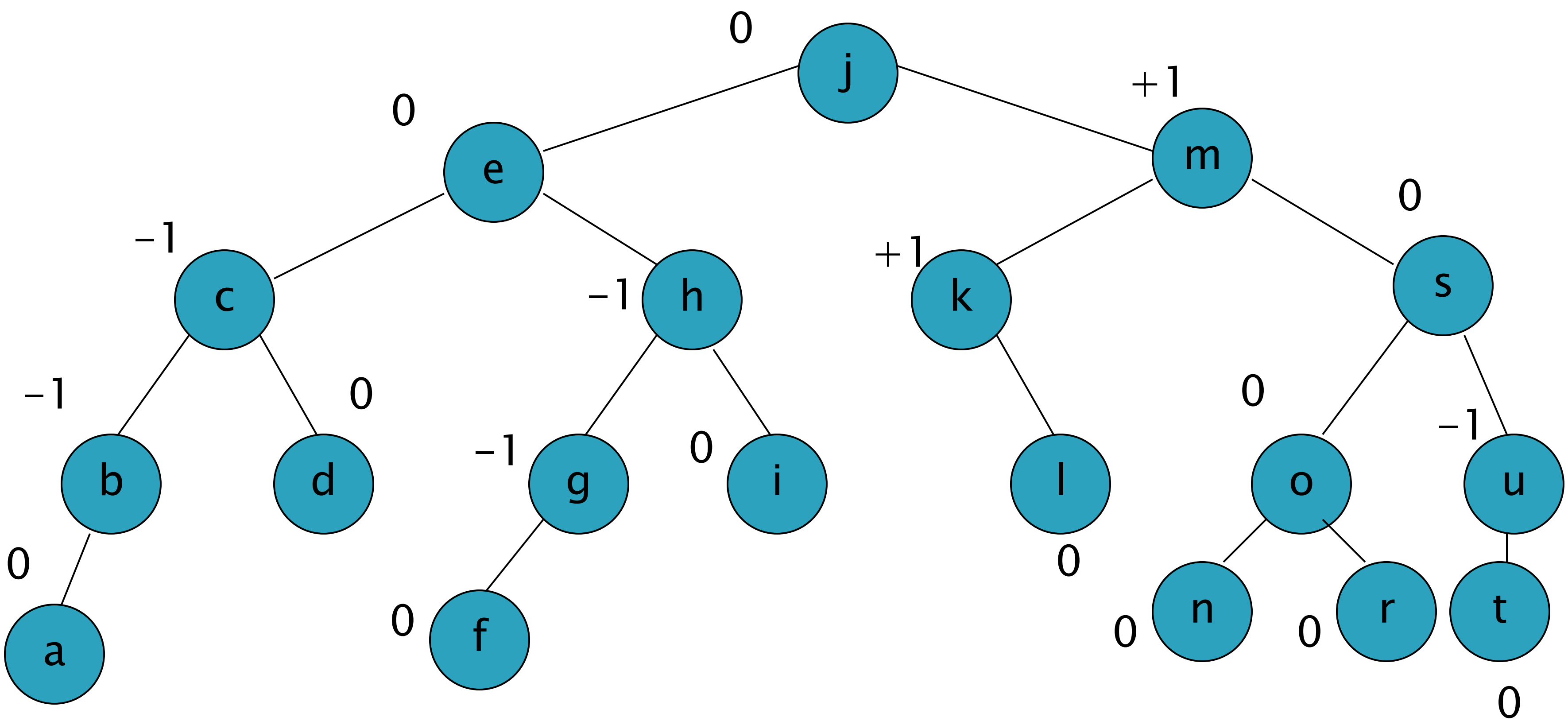


Now check again (case 3)
Right-Left Rotation (Double)

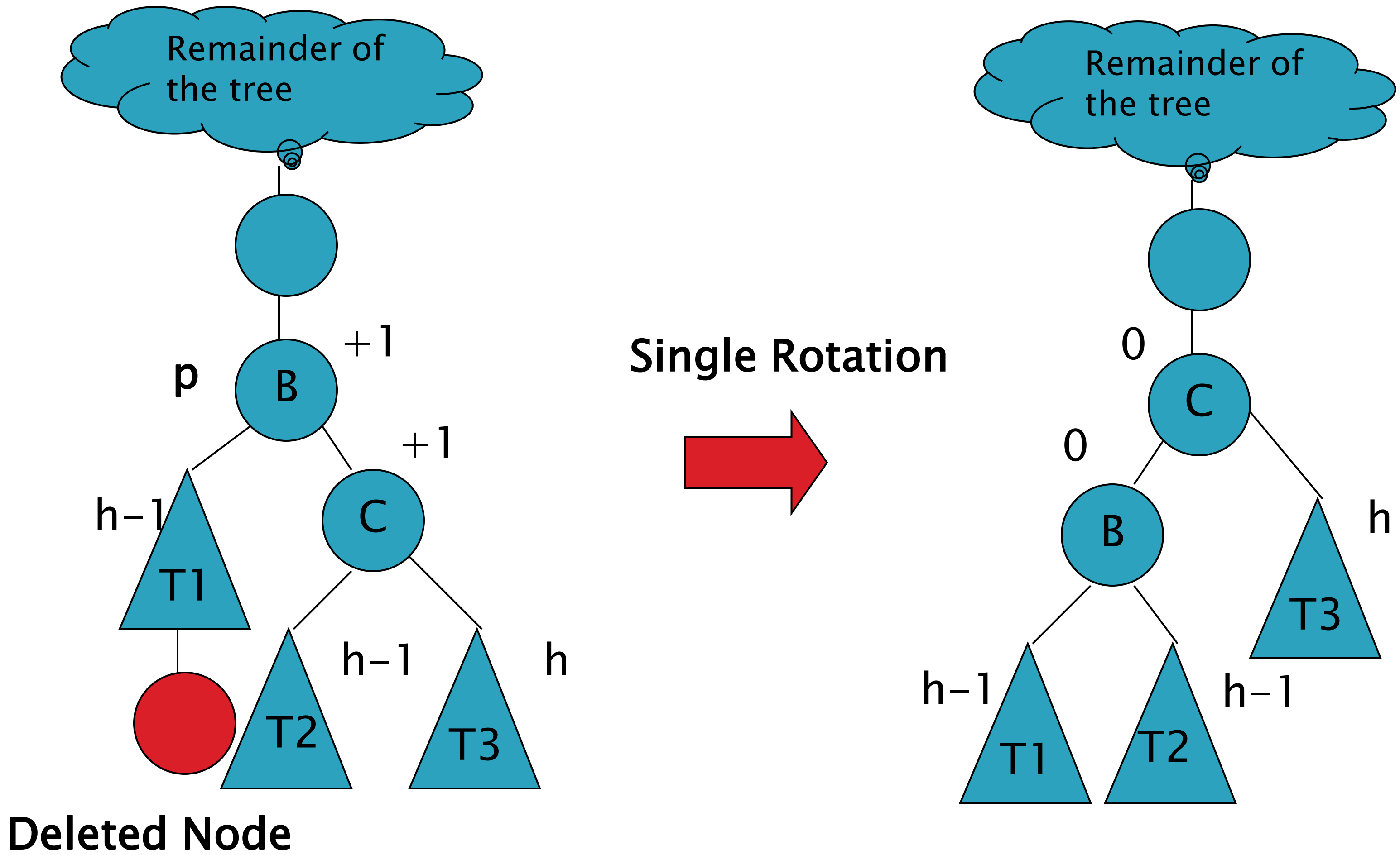
AVL Tree: Delete (Case 3)



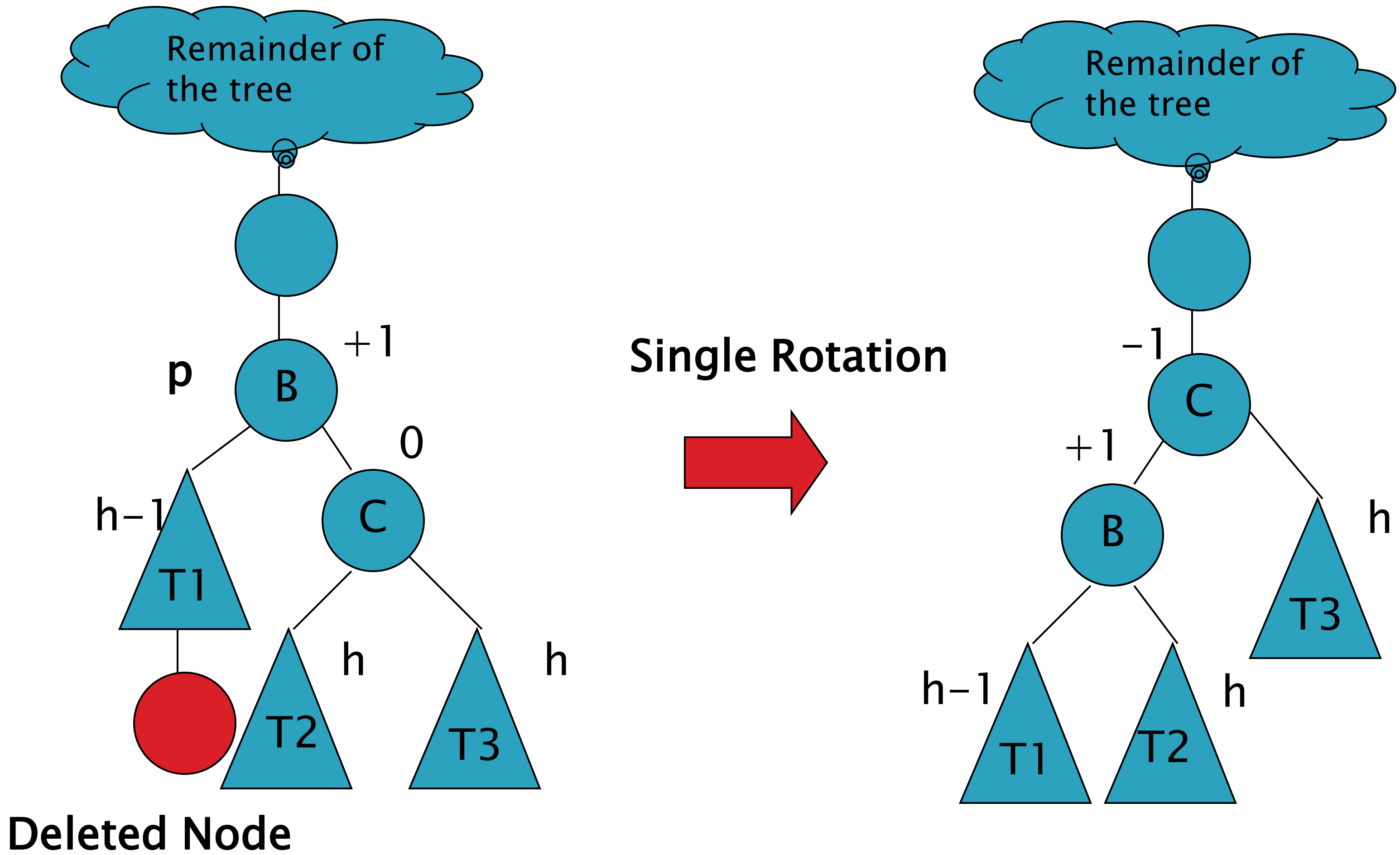
AVL Tree: Delete (Case 3)



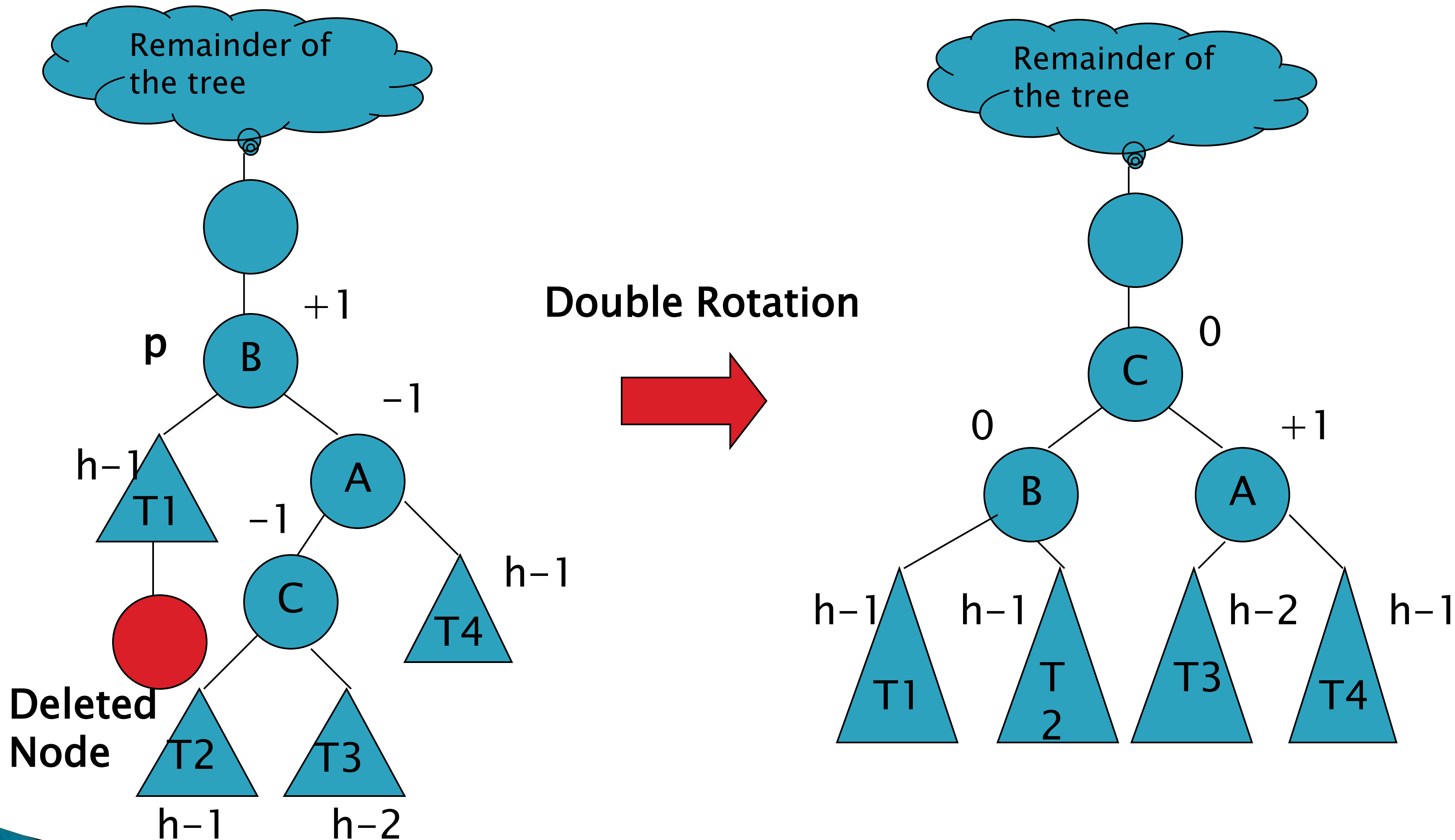
AVL Tree: Delete (Case 3: Sub-Case 1)



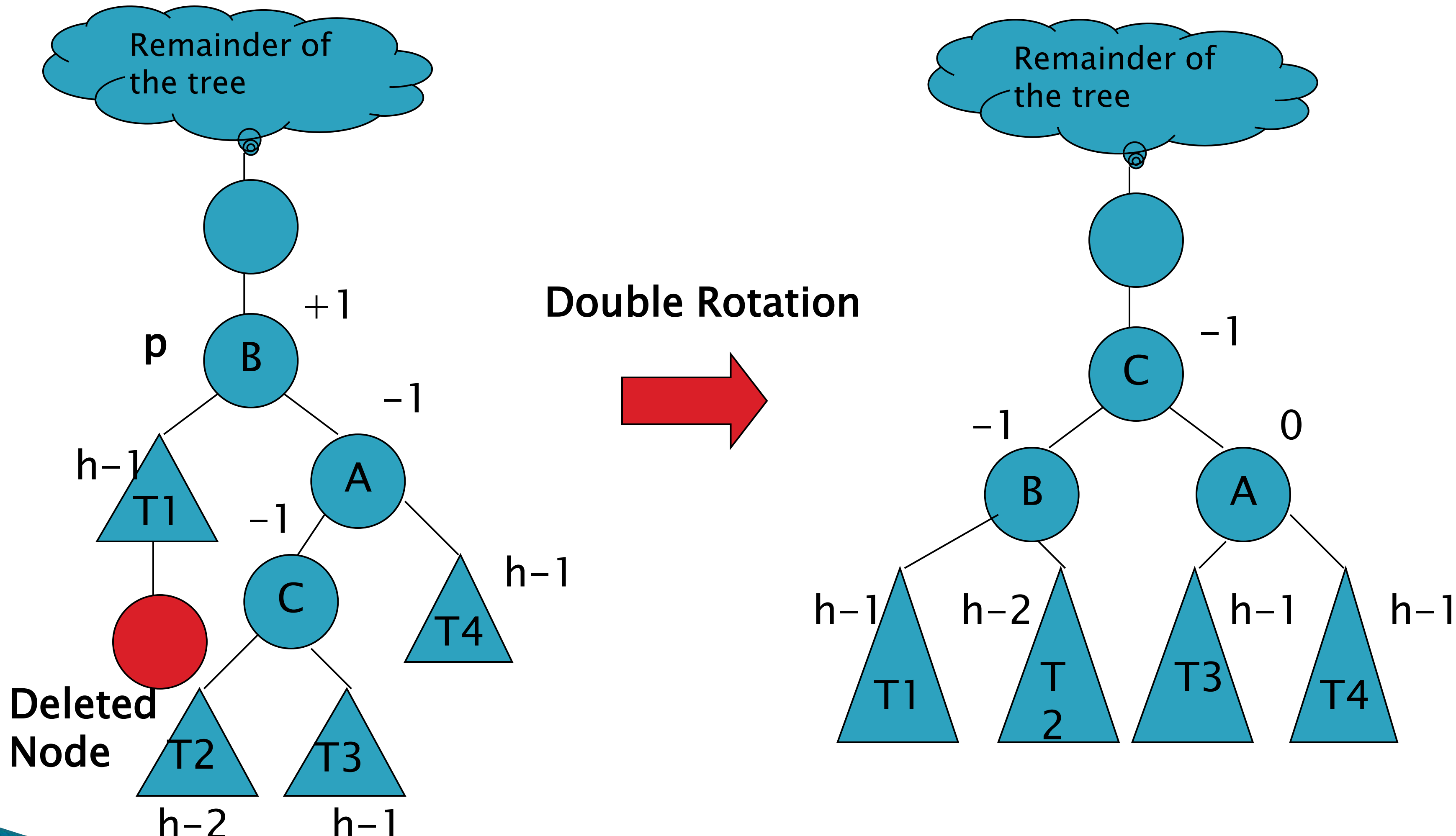
AVL Tree: Delete (Case 3: Sub-Case 2)



AVL Tree: Delete (Case 3: Sub-Case 3)



AVL Tree: Delete (Case 3: Sub-Case 4)



AVL Tree: Delete (Case 3: Other Sub-Cases)

- Sub-Case 5: mirror image of Sub-Case 1.
- Sub-Case 6: mirror image of Sub-Case 2.
- Sub-Case 7: mirror image of Sub-Case 3.
- Sub-Case 8: mirror image of Sub-Case 4.